UCD Smurfit Graduate School of Business

Data Mining

Assignment Three - Data Mining

Authors:
Conor Reid (16202630 - MSc B.A Part-time)

Lecturer:
Dr. Michael O'Neill

April 17, 2017

Classification and Clustering- Q1

Classification and clustering are two very common data exploration and interrogation techniques, but used to achieve different goals.

In general, classification involves a number of predefined classes (to which observations are assigned) and the goal is to take a new observation and predict which of these classes it belongs to. In this way classification is a type of supervised learning. A training dataset is required, consisting of training examples. Each example is a pair, made up of the input vector (a sum of each feature in the observation) and the signal (or label, the class the observation belongs to). Classification algorithms take these training examples and infer a mapping function, which is used for new and unseen examples.

On the other hand, clustering belongs to the family of unsupervised learning. Here, a mapping function is still inferred as above, however this time observations are unlabelled. We do not know which class observations belong to, or even how may classes there may be in the first place. There is thus no way to evaluate the accuracy of the results. Clustering then attempts to group observations in such a way that observations within each group are more similar to each other that those in other groups. Ideally these groups would be highly distinct. That is, spatially very separate when plotted.

R Code Analysis - Q2

The R code being analysed here is shown below. In general, this code aims to take some dataset and partition it into training and test subsets for model learning, and testing respectively.

```
> set.seed(1234)
> dataPartition <- sample(2,nrow(data),replace=TRUE,prob=c(0.7,0.3))
> trainData <- data [dataPartition ==1,]
> testData <- data [dataPartition ==2,]
```

Since there is a random element to the above procedure, a seed is set. This contains the randomness, and allows the same results to be obtained each time the code is ran. Changing the seed changes the results.

The `sample()` function in R takes a sample of a given size, with or without replacement. Here, `sample(2)` returns the numbers 1 and 2 in a random order. Passing in the data dataset, we can assign one of the two "labels" to each observation, thus designating which of the training (1) or test (2) datasets we want that observation to be in. `replace=TRUE` means that we are sampling with replacement. `sample(2, replace=TRUE)` may return "2 , 2" or "1, 2" etc.

The `prob(...)` term at the end indicates the split of the data between the two subsets. 70% goes to training, 30% to test. The following two lines simply pick out observations in the data and assigns them to each subset.

The Weka implementation of C4.5 algorithm - Q3

The C4.5 algorithm is used in generating decision trees used for classification, and is an extension the earlier ID3 algorithm, of the same authorship. The "M" parameter sets the minimum number of instances per leaf. That is, it dictates the degree of splitting at each node and thus effects the resulting accuracy of the results. For this reason a number of iterations of the algorithm is often run, with differing values of M. The model that results in the best accuracy is then chosen.

The `churnTrain` Dataset - Q4, Q5

The `churnTrain` dataset was analysed as fulfilment of this question. Upon inspection, it immediately became clear that the dataset was highly unbalanced. The dependant variable, *churn*, contains roughly 85.5% *"no"* values with the remainder being *"yes"* values. It is likely then that any model that is built on this data as-is will be more able to accurately predict the *"no"* class since it has more observations to base rules on.

Ideally one would use some balancing function to resample the minority class (*"yes"*) and increase predictive power here. One such function is `SMOTE`.

Results - Non-Pruned Classification Tree
$TreeSize = 32$
$Errors = 130(3.9\%)$

Confusion Matrix

	Yes	No	Total
Yes	369	114	483
No	16	2834	2850
Total	385	2948	3333

```
treeModel = C5.0(x = churnTrain[, -20], y = churnTrain$churn,
                 control = C5.0Control(noGlobalPruning=TRUE))
treeModel
summary(treeModel)
```

The above results relate to a classification tree that was built without a pruning step. The tree size is 32, and a relatively small error rate of 3.9%. The confusion matrix confirms that 130 instances were misclassified, being identified as *"no"* cases when they are actually *"yes"* or visa versa.

```
Attribute usage:

100.00% total_day_minutes
 93.67% number_customer_service_calls
 87.73% international_plan
 20.73% total_eve_charge
 11.34% voice_mail_plan
  8.01% total_intl_calls
  6.48% total_intl_minutes
  6.33% total_night_minutes
  5.34% total_eve_minutes
  0.57% total_eve_calls
  0.51% total_night_charge
  0.18% account_length
  0.18% total_day_charge
```

The attributes of the dataset that are used in constructing the tree are listed above. *total_day_minutes* is shown to be most important in determining churn, followed by 12 more variables in decreasing order of importance. There are 20 variables in the dataset, and so since 13 are used in building the model, it is quite dependant on the data.

Results - Pruned Classification Tree
$TreeSize = 27$
$Errors = 136(4.1\%)$

Confusion Matrix

	Yes	No	Total
Yes	365	118	483
No	18	2832	2850
Total	383	2950	3333

```
treeModel = C5.0(x = churnTrain[, -20], y = churnTrain$churn,
                 control = C5.0Control(noGlobalPruning=FALSE))
treeModel
summary(treeModel)
```

The results above relate to a classification tree that was built using a final pruning step, controlled by the $C50$ R package. The goal of pruning is to reduce the size of the tree, reducing the complexity of the final classifier. This is done by removing sections of the tree that contribute little predict power. Predictive accuracy should improve by reducing overfitting.

The error rate has shown to slightly increase here, by 0.2% which amounts to an extra 6 misclassified observations. This is very small, and the reduced dependancy of the model on the underlying data means that generalisability is greatly improved.

```
Attribute usage:

100.00% total_day_minutes
 93.67% number_customer_service_calls
 87.73% international_plan
 20.73% total_eve_charge
  8.97% voice_mail_plan
  8.01% total_intl_calls
  6.48% total_intl_minutes
  6.33% total_night_minutes
  4.74% total_eve_minutes
  0.57% total_eve_calls
  0.18% account_length
  0.18% total_day_charge
```

Again, the attributes most important to the pruned decision tree model are listed above in decreasing order of importance. This output is similar to that seen before for the un-pruned tree, however here we have 12 variables total which represents a reduction by 1 over the un-pruned tree. This may not seem significant but any reduction in model dependancy is certainly advantageous, especially when model performance is not greatly effected as is the case here.

The `GermanCredit` dataset - Q6

The `GermanCredit` dataset was analysed and modelled in a similar way to the previous question, involving some preliminary exploration followed by the growing of a classification decision tree. The following code was used to begin with.

```
library(caret)
data(GermanCredit)
table(GermanCredit$Class) #unbalanced; 300 - 700
```

The data is read in, and a table constructed of the explanatory variable (in this case, the *Class* variable which takes either a *Good* or a *Bad* value. It is apparent that this is quite unbalanced, with 300 *Bad* cases and 700 *Good* cases. As discussed previously, an imbalance like this in the dataset allows the majority class to be accurately modelled, but makes it difficult to pick up the rules that govern the minority class. In many cases, it is the minor class that is actually of interest, where one wants to be able to detect what is known as a *rare event*. The imbalance here probably wouldn't be too detrimental to model building, but it can be assumed that interested parties are more motivated to find instances of *Bad* credit than *Good* and act on that information.

For this reason it was decided to use the `SMOTE` function in R to balance the classes. `SMOTE` acts to artificially generate new instances of the minority class by using the nearest neighbours of these cases. In addition, majority instances are under-sampled resulting in a more balanced dataset. The code below was used to achieve this.

```
GermanCredit$Class <- as.factor(GermanCredit$Class)
GermanCredit <- SMOTE(Class ~ ., GermanCredit, perc.over = 100, perc.under=200)
GermanCredit$Class <- as.numeric(GermanCredit$Class)
table(GermanCredit$Class) #balanced; 600 - 600
dim(GermanCredit) #[1] 1200 62
```

With a balanced dataset, we can continue and build the now familiar classification decision tree. First we split the dataset into training and test subsets, that will be used to create the tree and test its performance respectively. The code below is used to do this.

```
#dataPartition <- sample(2,nrow(GermanCredit),replace=TRUE,prob=c(0.5,0.5))
dataPartition <- sample(2,nrow(GermanCredit),replace=TRUE,prob=c(0.7,0.3))
trainData <- GermanCredit[dataPartition ==1,]
testData <- GermanCredit[dataPartition ==2,]
trainData$Class <- as.factor(trainData$Class)
dim(trainData) #[1] 864 62
dim(testData) #[1] 336 62
```

A split of 70% for test and 30% is primarily used, although a 50% split each way was also tested. The former is a very common split used in literature, but the results of both will be discussed further on. Our primary split is used to create each dataset, which have dimensions as shown.

We are now ready to fit the model. The code below is used to do this.

```
treeModelPruned = C5.0(x = trainData[, -10], y = trainData$Class,
                  control = C5.0Control(noGlobalPruning=FALSE))
treeModelPruned
summary(treeModelPruned)
```

The results on the training dataset are outlined below.

$Treesize = 67$
$Errors = 44(5.1\%)$

Confusion Matrix

	Good	Bad	Total
Good	393	34	427
Bad	10	427	437
Total	403	461	864

The above results can be considered good, with a low error rate and by and large the correct allocation of *Good* and *Bad* labeled instances into their respective groups. We now look at the results when we used our test data on this model.

```
predictions <- predict(treeModelPruned, testData, type="class")
table(testData$Class, predictions)
```

The below confusion matrix is achieved.

Confusion Matrix

	Good	Bad	Total
Good	134	39	173
Bad	19	144	163
Total	153	183	336

This confusion matrix can be used to find the sensitivity (proportion of positive instances correctly identified) and the specificity (proportion of negative instances correctly identified). They are calculated as:

$$Sensitivity = \frac{TruePositives}{TruePositives + FalseNegatives} = \frac{134}{134 + 39} = 77\% \tag{1}$$

and

$$Specificity = \frac{TrueNegatives}{TrueNegatives + FalsePositives} = \frac{144}{144 + 19} = 88\% \tag{2}$$

These could be considered good results from our training set, albeit worse than that seen in the training set results. This may suggest overfitting at the training stage. To counter act this, an alternative split of 50% each way was used.The resulting using the same methodology on this split are shown below.

$$Sensitivity = \frac{236}{236 + 61} = 79\% \tag{3}$$

and

$$Specificity = \frac{246}{246 + 63} = 80\% \tag{4}$$

This split reduced the tree to a size of 47 which is significantly less than the previous result. It can be said then that the model dependancy is less, and it should generalise better. The sensitivity and specificity above are interesting results; out ability to detect *Good* instances has grown, while our ability to detect *Bad* instances has reduced. In the context of our problem, it may be that the first model is better since we can have greater confidence when we find a *Bad* instance that it is in fact just that. Further analysis of the problem may further inform which model is better.

The `college` dataset

Q1 & Q2 - What can we learn from the dataset, and classification techniques used.

One of the most obvious features of the dataset is that it is highly unbalanced, with almost 90% of colleges falling into the "Not-Elite" class. This is to be expected since in the real-world we recognise a small percentage of academic institutes as elite performers. This unbalance will need to be accounted for in any subsequent analysis.

```
college = read.csv(file="data/college.csv",head=TRUE,sep=",")
college = college[,-c(1,2)] #dont need the row num, or college name (all are unique)
table(college$isElite)
>> Elite  Not Elite
>> 78     699
```

We can perform an aggregate across features in this dataset and split by elite status, using the code below. This gives a good indication of how useful each feature may be for dividing the classes. Mean values across each feature seem to differ substantially with elite status, with the average graduation rate 20% higher for elite colleges for example. From this preliminary finding, it is likely that an accurate classification model is achievable that will be used to classify new, unseen instances to good effect.

```
aggregate(college[, 1:4], list(college$isElite), mean)#averages split by elite or not
>>   Group.1  accept_rate  Outstate  Enroll  Grad.Rate
>> 1   Elite    0.5434093  15248.564  1060.7179  83.38462
>> 2 Not Elite 0.7696379  9904.166   748.6452   63.46352
```

Figure 1 shows the acceptance rate density per elite status. It is clear from the analysis above that the acceptance rate is very much higher in non-elite colleges, the plot below is simply a visual confirmation of this. Elite colleges show potentially two clusters of acceptance rate density, one at approximately 45% and another at approximately 80%. It may be worthwhile exploring this further later in this analysis. Non-Elite colleges have a distinctly normal, although skewed to the right, distribution with most colleges admitting roughly 80% of applicants.

```
qplot(accept_rate, data=college, geom="density", fill=isElite, alpha=I(.5),
     main="Distribution of Acceptance Rate by Status", xlab="Accept Rate",
     ylab="Density")
```

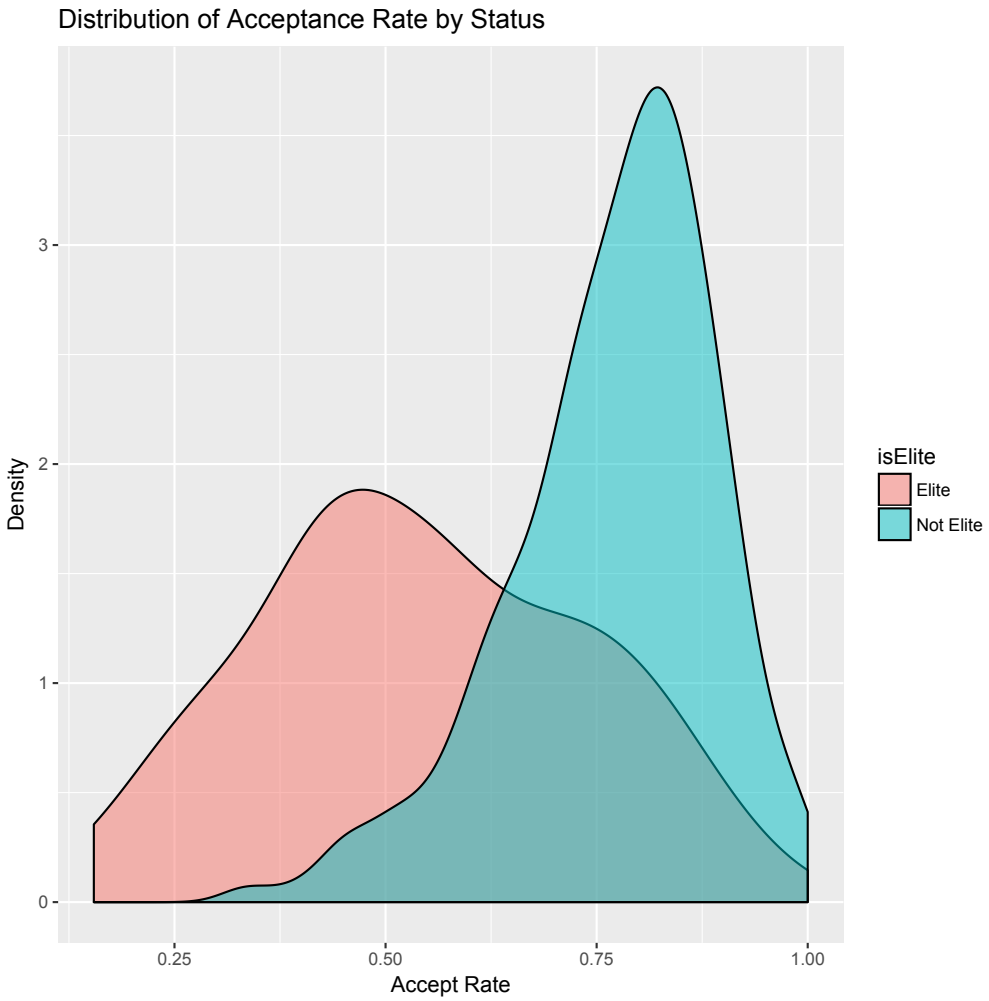

Figure 1: Acceptance Rate by Elite Status

For the final piece of exploratory analysis, we look to see how the "Private" status of the college may be influencing its Elite status. Approximately 72% of Non-Elite colleges are private, compared to approximately 83% of Elite schools. This points to a potential importance associated with the "Private" feature for separating out classes of colleges.

```
table(college$isElite,college$Private)
>>            No     Yes
>> Elite      13     65
>> Not Elite  199    500
```

A classification model can now be built using this dataset. As mentioned above, the dependant variable (i.e. the Elite status of the college) is highly unbalanced, and so the code below is used to resample the minority class and under-sample the majority class to yield a more balanced dataset. This will facilitate a model that is better able to detect instances from the minor class (the Elite colleges) as well as the majority

class (the Non Elite colleges).

```
college$isElite = as.numeric(college$isElite)
college$Private = as.numeric(college$Private)
college$isElite = as.factor(college$isElite)
collegeBalanced = SMOTE(isElite ~ ., college, perc.over = 300, perc.under=300)
collegeBalanced$isElite = as.numeric(collegeBalanced$isElite)
table(collegeBalanced$isElite)
>> 1    2
>> 312  702
```

The SMOTE function is used to achieve the over and under-sampling. The key parameters here are the perc.over parameter (which represents a number that drives the decision of how many extra observations from the minority class are generated) and perc.under (which represents a number the drives the decision of how many extra observations from the minority classes are selected for each case generated from the minority class). We now have a more balanced dataset, albeit still one-sided towards Non Elite colleges (class 2). With more time, an analysis of how to best balance this dataset would be desirable.

The dataset must now be split into training and test subsets, which are used to build the model and test it respectively. A 70:30 split is used, as is standard practice. The code below is used to do this.

```
dataPartition = sample(2,nrow(collegeBalanced),replace=TRUE,prob=c(0.7,0.3))
trainData = collegeBalanced[dataPartition ==1,]
testData = collegeBalanced[dataPartition ==2,]
trainData$isElite <- as.factor(trainData$isElite)
dim(trainData)
>> [1] 726 6
dim(testData)
>> [1] 288 6
```

The classification models can now be built. Included first is a discussion of the algorithm usage, a discussion of results across each algorithm follows (although the confusion matrix or each algorithm test run is included, as well as the percentage accuracy) . We begin by building a classification tree as is used throughout this report. The code used to achieve this is below.

```
treeModelPruned = C5.0(x = trainData[, -6], y = trainData$isElite,
                  control = C5.0Control(noGlobalPruning=FALSE))
treeModelPruned #for output, see attached R script
summary(treeModelPruned) #for output, see attached R script
predictions <- predict(treeModelPruned, testData, type="class")
table(testData$isElite, predictions)
>>   predictions
>>    1  2
>> 1  74 20
>> 2  12 182
mean(testData$isElite == predictions)
>>[1] 0.8888889
```

The C5.0 function is used. This takes the dataset as the first parameter, minus the target variable which is in position six in the above example. That target variable is passed in second, along with a parameter to allow pruning to take place. This is done in order to build a simplified tree that will be easier to interpret.

Next, the random forest model is used on the same training and test dataset. Also known as "random decision trees", this is a method of constructing a multitude of decision trees at training time and outputting the class that most frequently occurs across all trees (the mode class). It is known that random trees help address the overfitting issue of normal classification trees, and so should provide better results in this case. The code used to build the random forest is below.

```
library(randomForest)
randomForestModel = randomForest(isElite ~ ., data=trainData, ntree=100, proximity=T
    )
table(predict(randomForestModel), trainData$isElite)
```

```
print(randomForestModel)
plot(randomForestModel, main= "")
importance(randomForestModel)
randomForestModelPred = predict (randomForestModel, newdata = testData)
table(testData$isElite,randomForestModelPred)
>>    1  2
>> 1  77 17
>> 2  12 182
mean(testData$isElite == randomForestModelPred)
>> [1] 0.8993056
```

The `randomForest` function is used to build the decision trees. This takes the target variable as the first parameter, and the dataset as the second. The `ntree` term decides the number of trees to grow. It is desirable to pick an amount such that the error is stabilised, but not too many so as to cause overfitting. From some light testing, it is apparent that increasing or decreasing this number makes little difference. Again, with more time an optimal number here can be found. The final `proximity` term decides whether the proximity measure (i.e. the "closeness" or "nearness" between pairs of observations) among the rows is calculated. The proximity is often used in replacing missing data and locating outliers, and is set to "True" here.

The third and final classification model that is used is quadratic discriminant analysis (QDA). This method is similar to linear discriminant analysis (LDA), but instead of a linear separator it uses a quadric surface. That is, it works to create a surface that separates the classes in space. When new observations are introduced, they are effectively plotted to the same space and their class then is determined by which side of the dividing surface they lie on. Since this is simply a more general version of LDA, it is easily implemented in code which is shown below.

```
library(MASS)
qdaModel = qda(formula=isElite~., data=trainData)
summary(qdaModel)
sqdPred = predict(qdaModel, newdata=testData)
table(actual=testData$isElite, predicted=sqdPred$class)
>>    1  2
>> 1  68 26
>> 2  17 177
mean(sqdPred$class == testData$isElite)
>> [1] 0.8506944
```

Overall, it is clear that all three models perform very well and achieve high amounts of class matches at test time. Classification decision trees correctly allocated an Elite status approximately 89% of the time, with the random forest technique managing a 1% increase to approximately 90% accuracy. Quadratic discriminant analysis achieved an accuracy score of just 85% making it the least powerful classification technique in this instance.

Both decision tree algorithms allow the study of the important variables in their respective constructions. Important variables are those that can be best used to split the data into classes. The pruned decision tree model showed that both the "accept_rate" and "Outstate" variables were highly important, with attribute usage scores of 100%. This means those variables are used in classifying every single observation in the dataset. The "Private" and "Grad.Rate" variables follow in importance, at 72% and 40% respectively. These are less important, but still highly useful. The only variable shown to have little correlation with Elite status is the "Enroll" variable.

The random forest algorithm agreed with some of these findings. It found the "accept_rate" and "Outstate" features to be the most important, followed at a distance by "Grad.Rate". Interestingly however, is that this model found that "Enroll" was the next useful feature in predicting Elite status with a non-negligible impact. "Private" , which proved to be highly important above, was the least powerful here with negligible impact.

It is clear then that the acceptance rate and out-of-state tuition features have high predictive power for determining the Elite status of a college. It could also be said that the graduation rate is also useful here. The role of the private status of the college, and the amount of enrolled students is yet unclear. Fur-

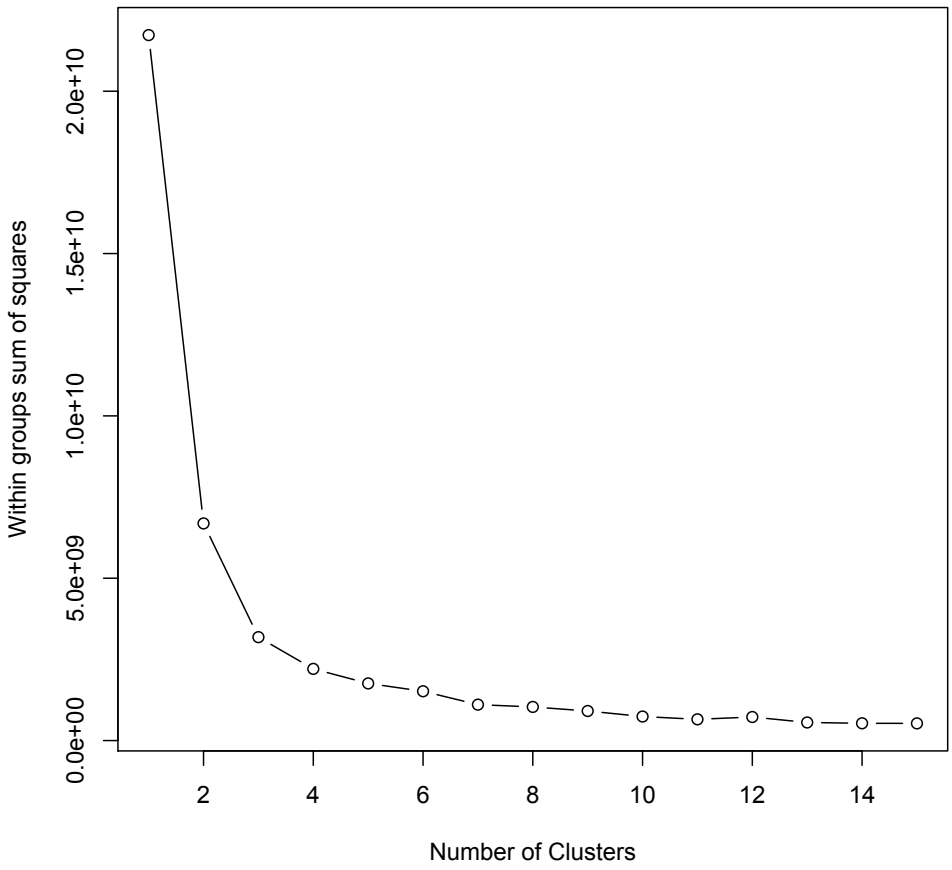

Figure 2: Number of Clusters and Within group SOS

ther analysis may be required to satisfy this, potentially using a bigger dataset or employing other algorithms.

Q3 - Clustering analysis

The final part of this analysis involves removing the target variable, the Elite status, from the college dataset considered above and carrying out clustering on the remaining features. Up until this point, we have been considering supervised learning, where we know which class observations belong to before building the model. By removing this class feature, we move into the unsupervised learning space where we try to group observations into as-yet-unknown classes. There is potential benefit here in doing so, since we can validate the decision to use a binary class system in the first place but also to see whether the Elite status of colleges should be broken down further.

The code used to facilitate this analysis begins as below;

```
collegeBalancedClustering = collegeBalanced[,-6]
wss = (nrow(collegeBalancedClustering)-1)*sum(apply(collegeBalancedClustering,2,var))
for (i in 2:15) wss[i] = sum(kmeans(collegeBalancedClustering, centers=i)$withinss)
plot(1:15, wss, type="b", xlab="Number of Clusters",
    ylab="Within groups sum of squares")
```

First the isElite variable is removed from the dataset. Next, a plot is generated that maps the hypothetical number of clusters used against the within group sum of squares (figure 2 . Ideally this measure would be small, since that would indicate the presence of tightly clustered distinct centres. The plot below shows that using 3 clusters yields a low measure. Using more clusters yields practically no advantage.

For this reason then a clustering model will be built using three clusters. The code below is used to achieve this.

```
kmeansCluster = kmeans(collegeBalancedClustering, 3) # 3 cluster solution
aggregate(collegeBalancedClustering,by=list(kmeansCluster$cluster),FUN=mean)
>> Group.1   accept_rate  Outstate    Enroll  Grad.Rate   Private
>> 1     1    0.7735830  11752.756  568.2968  71.58158  1.915789
>> 2     2    0.5555162  17662.161  793.0853  84.03460   1.985348
>> 3     3    0.7468155  6770.816   1234.6934  57.40017   1.365651
```

The aggregate function is used to find the mean value across all features split by cluster. This allows a preliminary look into how well defined each cluster is. Ideally the means would be far apart, and this trend is seen in places above. For each variable there is at least one cluster whose mean deviates from the others significantly, with the cluster that deviates being different from feature to feature. This indicates that perhaps a three state target variable is more appropriate rather than the binary "isElite" seen in the original dataset.

Having determined that there may be subgroups within either Elite or Non-Elite colleges, it would be interesting to see where these subgroups by be. An attempt is made to do this by merging back in the isElite variable from the original dataset and looking at which classes are "mismatched" (i.e. records assigned to the three class we have potentially discovered.)

```
collegeBalancedClustering = data.frame(collegeBalancedClustering,
   kmeansCluster$cluster)
collegeBalancedClustering$rowNames = row.names(collegeBalancedClustering)
collegeBalanced$rowNames = row.names(collegeBalanced)
actualResults = collegeBalanced[,c(6,7)]
collegeBalancedClustering = merge(collegeBalancedClustering, actualResults,
   by="rowNames" )
```

```
library(sqldf)
#matches
sqldf('select count(*) from collegeBalancedClustering
            where "kmeansCluster.cluster" = isElite')
#nonmatches
sqldf('select count(*) from collegeBalancedClustering
            where "kmeansCluster.cluster" != isElite')
#extraGroup
sqldf('select count(*) from collegeBalancedClustering
            where "kmeansCluster.cluster" = 3')


#are those observations marked 3 by the clustering actaully 1's or 2's in the
#orginal dataset?
sqldf('select isElite, count(*) from collegeBalancedClustering
          where "kmeansCluster.cluster" = 3
          group by isElite')
```