Classification of variables, X1 and X2, using different classification methods and optimisation techniques

Nicholas P.J. Harper, 700038738

Introduction

Variables X1 and X2 are simulated data of 1000 datapoints that have had some noise or random error added and can be found in the Classification.csv dataset. These data have been split into 2 groups, 0 and 1, and a series of classification methods will be used in this study together with different optimisation techniques, which are given in Table 1 below:

Classification method	Optimisation technique
Linear Discrimination Analysis (LDA)	Receiver-Operator Curve (ROC)
Quadratic Discrimination Analysis (QDA)	Receiver-Operator Curve (ROC)
Logistic regression	Receiver-Operator Curve (ROC) and Confusion matrix
Support Vector Machines (SVM) - Linear	Cross-validation and Confusion matrix
Support Vector Machines (SVM) – Radial	Cross-validation and Confusion matrix
Support Vector Machines (SVM) - Polynomial	Cross-validation and Confusion matrix
K-nearest neighbours (KNN)	Cross-validation and Confusion matrix
Decision trees (bonus)	Confusion matrix
Random forests (bonus)	Confusion matrix

Table 1: The series of classification methods will be used in this study with their different optimisation techniques

The two variables, X1 and X2, are summarised in Table 2, plotted against each other in Figure 1 as well as in pairs plots with the Group class in Figure 2. The means and medians of X1 and X2 are found to be close to 0, with an inter-quartile range of (-0.67, 0.67). Also, the group classes, 0 and 1, for variables X1 and X2 overlap each other, whilst there is a good scatter of datapoints in the plot of X1 against X2.

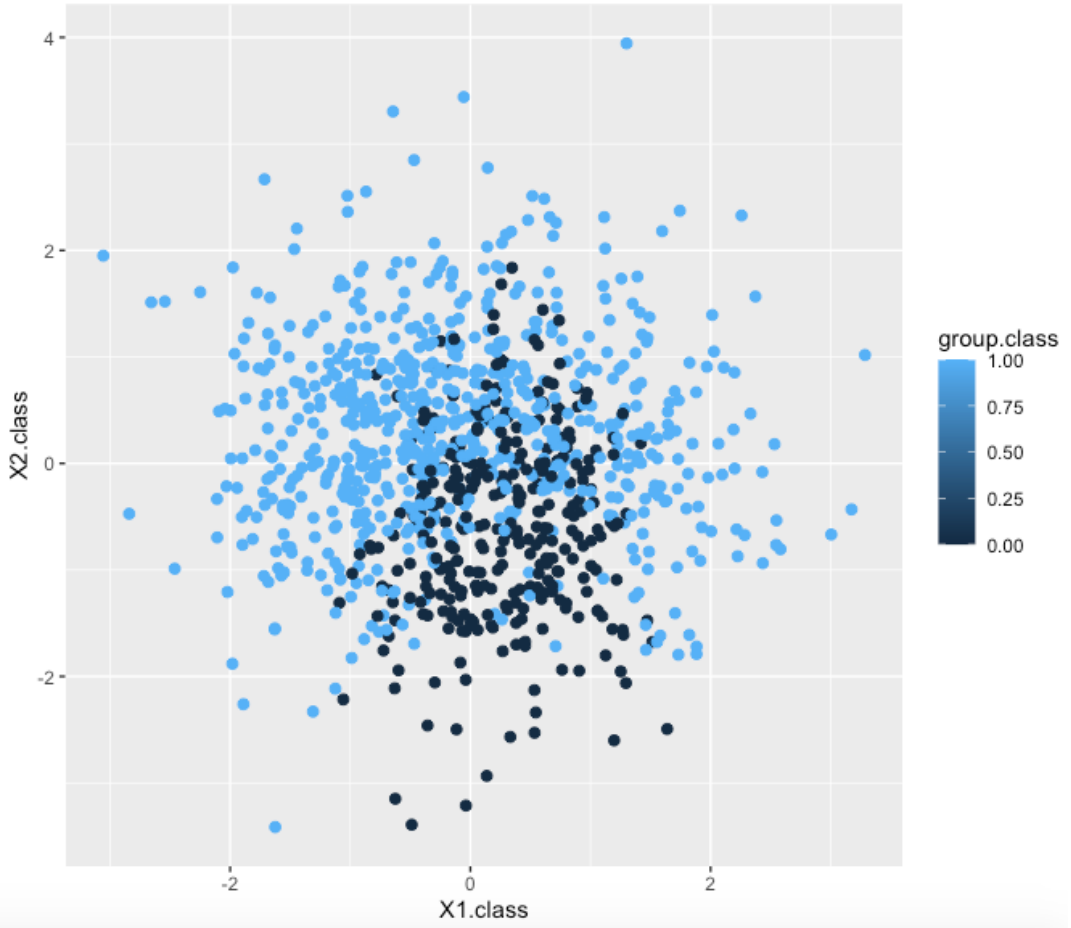

Figure 1: Coloured ggplot by group of X1 against X2, showing overlapping groups for each variable

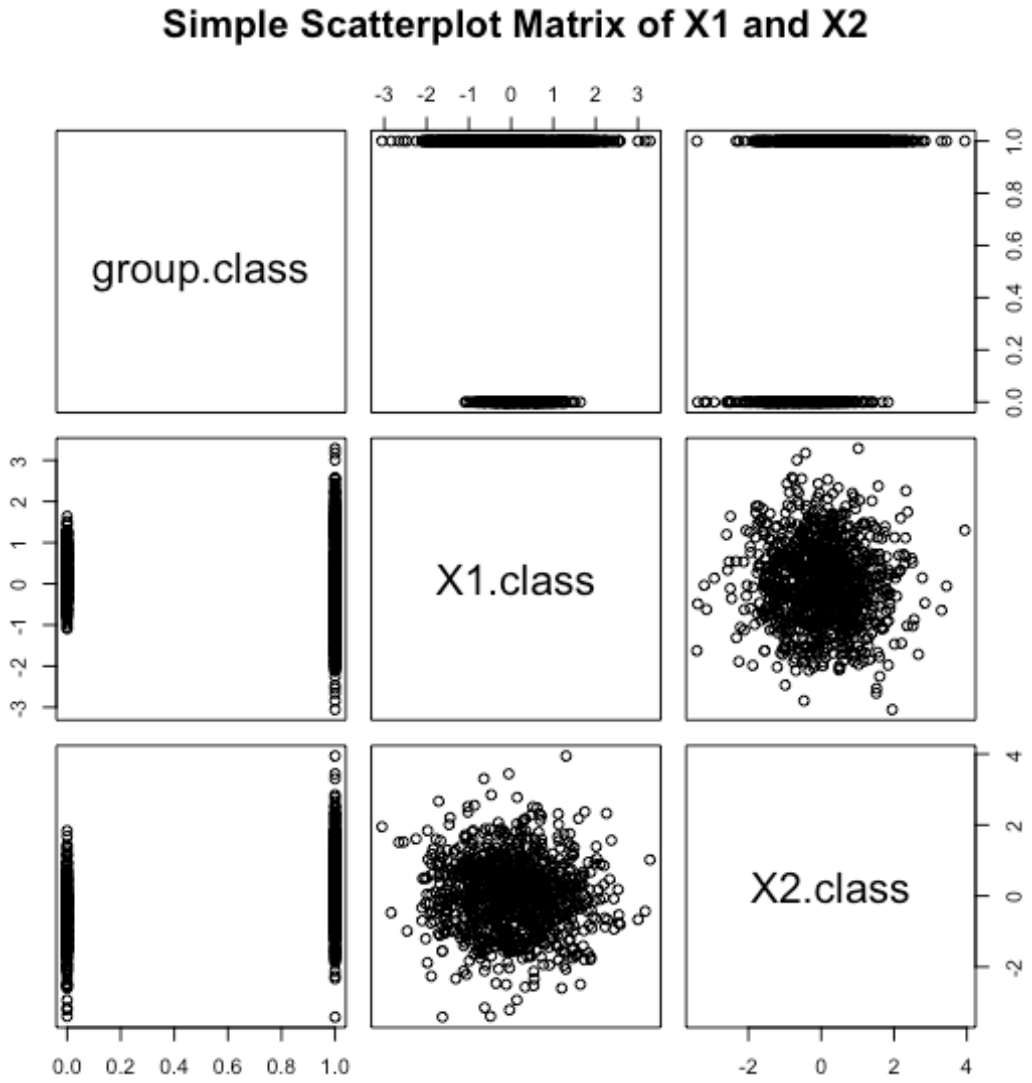

Figure 2: Pairs plot of group class, X1 and X2, showing overlapping groups for each variable and a good scatter when comparing the variables

X1	X2	Group
Min. :-3.055858	Min. :-3.41429	Min. :0.000
1st Qu.:-0.679636	1st Qu.:-0.62566	1st Qu.:0.000
Median :-0.038071	Median : 0.01472	Median :1.000
Mean : 0.002033	Mean : 0.01631	Mean :0.682
3rd Qu.: 0.649037	3rd Qu.: 0.66946	3rd Qu.:1.000
Max. : 3.285469	Max. : 3.94396	Max. :1.000

Table 2: Summary statistics for the variables, X1 and X2, as well as the Group class

Methods for classification and optimisation

In order to analyse the groups that each datapoint should go into, the dataset needs to be split into a training set, called “train.class”, which comprises a random selection of 80% of all the data, and a testing test, called “test.class”, which comprises the other 20% of the data. With these training and testing sets in place, the different classification

methods and optimisation techniques can be employed to determine with produces the most accurate results. The methods for each classification method and optimisation technique are provided here:

Each method below uses a different technique to predict the class of each datapoint so that it can be compared to the true class value. Each method attempts to approximate to the Bayes classifier, which assigns an observation to its true class, produces the lowest possible error rate and assumes that observations are independent of each other. The results from the methods are compared with each other and then each with another dataset containing the true values for X1, X2 and their associated classes.

Linear and Quadratic Discriminant Analyses (LDA and QDA)

LDA and QDA are supervised classification methods that can be used to approximate a Bayes classifier by putting estimates for mean (μ_k), variance (σ^2), and proportion (π_k) of training observations belonging to a particular class. The methods involve separating different random variables (X) into different classes (j) using a probability density function ($f_i(X)$) that is associated with each class. This classification by discrimination means that if the random variable, X, has the highest probability (say p > 0.5) that it is in class j, then it is placed into class j. This is achieved via either a maximum likelihood rule (i.e. equal probability that each class can occur assumed) or Bayesian rule (i.e. use knowledge of class prior probabilities) to allocate each random variable, X, to class j, based on the argument:

$$j = \arg\max_i * \ f_i(X)$$

This means that the decision boundary between any 2 classes is where the probability of being in either class is equal (i.e. p = 0.5). Further differences between LDA and QDA are that:

- With LDA, all classes have the same covariance matrix (Σ) and a multivariate normal distribution, each class has its own mean vector (μ_k), (p+1) parameters are needed to build each discriminant function, (k-1) discriminant functions are needed (with the first one as the base) and the decision boundary between any 2 classes is a linear function of X. This function is given by:

$$\delta_{i(X)} = \ X^T * \Sigma^{-1} - \frac{1}{2} * (\mu_k)^T * \Sigma^{-1} * \mu_k + \log(\pi_k)$$

  Where: $\mu_k = \frac{1}{n_k} * \Sigma_{i:y_i=k} x_i$ (mean of training observations from kth class)
  $\pi_k = n_k/n$ (proportion of training observations belonging to kth class)
  And: Σ is the covariance matrix

- With QDA, each class have their own covariance matrix, so the means, covariances and class prior distributions need to be estimated for each discrimination function, and also the decision boundary between any 2 classes is a quadratic function of X. The function is given by:

$$\delta_{i(X)} = \ -\frac{1}{2} * (\log|\Sigma_k|) - \frac{1}{2} * (X - \mu_k)^T * \Sigma^{-1} * (X - \mu_k) + \log(\pi_k)$$

Optimisation by ROC curve

A Receiver-Operator (ROC) curve is often used for optimisation with LDA, QDA and logistic regression classification. It plots the predicted results against the observed results of True positive rate (or sensitivity) against False positive rate (or 1-specificity) and different ratios of these parameters are calculated at different "cutoff" thresholds (p) of classification. The formulae for True and False positive rates are given by:

$$True\ positive\ rate\ = \frac{True\ positive}{(True\ positive + False\ negative)}$$

$$False\ positive\ rate\ = \frac{False\ positive}{(False\ positive + True\ negative)}$$

To plot results of the ratio of these rates from each threshold (p) on an ROC curve, an algorithm called the Area Under the Curve (AUC) is employed, with results displayed as of a fraction between 0 and 1. This scaling measures the ranking of predictions and, as it shows all classification thresholds (0 < p < 1), the quality of model predictions can be assessed from the curve shape. Where predictions are 100% correct, this has an AUC value of 1, with a rule of thumb designed for AUC values < 1 in Table 3.

AUC value	Relative description of discriminatory power
0.9 – 1.0	Outstanding
0.8 – 0-9	Excellent
0.7 – 0.8	Acceptable
< 0.7	Poor

Table 3: Rule of thumb to describe relative Area Under the Curve (AUC) value from a Receiver-Operator (ROC) curve

The position of the "optimal" point for "cutoff" threshold (p) is close to the left-hand corner, but its exact position depends on whether we want either to minimise the false positives (i.e. SPAM filters) or to maximise the true positives (i.e. some medical diagnoses). Further, a high threshold often related to high specificity and low sensitivity and vice versa for low thresholds.

Logistic regression

This is a very good method for predicting qualitative (categorical) responses. For 2 classes, we can usefully think of binary discrete outputs, say 0 and 1, with a logistic regression line that forms an S-shaped, sigmoidal curve between the points. Binary classification using the logistic regression with a generalised linear model (glm) in R is achieved by the following steps:

1. Use the training data to estimate the coefficients $\beta 0$ and $\beta 1$ in the formula given next and fit a generalised linear model (glm) to the available data:
$$P(Y = 1|X) = e^{\beta_0 + \beta_1 * X} / (1 + e^{\beta_0 + \beta_1 * X})$$
2. Predict the class of an independent observation, X, which is saved in the test dataset, and choose a "cutoff" threshold, p*, which is commonly: p* = 0.5.
3. Use the estimated coefficients, $\beta 0$ and $\beta 1$, to calculate: p = P(Y=1|X) for each observation, X, for the probability that X belongs to class j=1.
4. Assign a class, Y, the observation, X, such that: Y=1, if p > p* = 0.5, and Y=0, otherwise.

Optimisation results by Confusion matrices

Confusion matrices are produced using the carat package as an additional optimisation technique for the K-nearest neighbours (KNN), support vector machines (SVM) and logistic regression (as well as decision trees and random forests) methods. It shows how well predictions and observations match, so if 0 = True and 1 = False in the matrix:

- True positive (top left-hand corner: Observation = 0; Prediction = 0)
- False negative (bottom left-hand corner: Observation = 0; Prediction = 1)
- False positive (top right-hand corner: Observation = 1; Prediction = 0)
- True negative (bottom right-hand corner: Observation = 1; Prediction = 1)

This optimisation technique runs an algorithm that measures the accuracy, sensitivity, specificity and other statistic results between a set of predictions and the testing set. In the case of classification, these are the class value predictions from the testing set and the observed classes in the testing set. The results can be easily compared between different classification methods to assess which method will be the preferred method.

Support Vector Machines (SVM)

This is another supervised learning method that assumes that in a 2-class scenario, both classes are linearly separable, so a line or hyperplane can be drawn with all points of one class on one side and all points of the other class on the other side. The logic in the SVM method puts each point into a class before calculating the maximum distance (C) for such an "optimally separating" line or hyperplane to be from each point. The decision boundary (or margin) for any 2 classes is then the hyperplane that lies in between the classes, which is drawn at the maximum distances (or margins) to the points in each class.

Also, points that are on the wrong side of the line or hyperplane must be as close to the line or hyperplane as possible. This is because to maximise the distance, C, the "Cost" parameter constant must be calculated, which is visualised in Figure 3 and is given by:

$$\frac{1}{C} * \Sigma_{i=1} \xi_i \leq \text{"Cost"}\ constant$$

Where: ξ_i= distance that the i^{th} point is on the wrong side of the margin

And: "Cost" = ratio of sum of distances for points on wrong side vs distances of points on right side

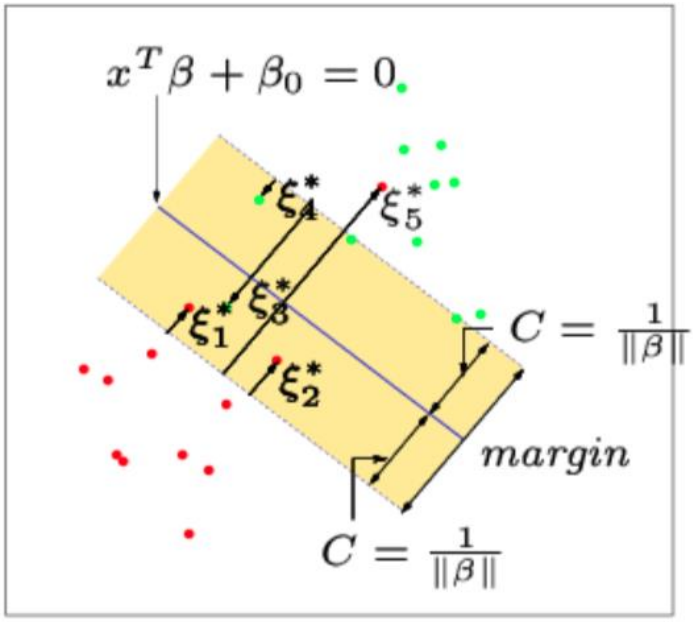

Figure 3: Schematic representation of how a Support Vector Machine works by drawing a line or hyperplane that is at a maximum distance from points in different classes, and using the "Cost" or "tuning" parameter to minimise the distance that a point on the wrong side of a line or hyperplane is from this margin

Optimisation by Kernel function transformation

This "cost" function can be optimised by transformations using different SVM Kernel functions. These functions are:

- Linear (i.e. Kernels that create linear hyperplanes)
- Radial (i.e. Kernels that create circular hyperplanes)
- Polynomial (i.e. Kernels that create hyperplanes with quadratic and higher terms)

The latter 2 techniques used in non-linear scenarios with SVM Kernel functions that transform complex point arrangements and places them into classes using predefined labels. The "optimal cost" can be displayed in a plot of accuracy against "Cost" (or the weight of how many samples are inside the best fitting margin between the classes) for each of the SVM techniques.

K-nearest neighbours (KNN)

This is a flexible method designed to estimate the Bayes Classifier, which assigns each observation to the most likely class given it predictor values (x_0). Consequently, this non-parametric method takes each datapoint and places it into a class using the Hamming distance from other points for classification (i.e. the number of changes that need to be made to transform a vector strong from the original form to the form required). Then, it determines whether the majority of those points are in class 1 or class 0 (for a 2-class scenario). For example, where there are 5 neighbouring points and 4 are in class j=1, then that point will go into class j=1. This is given by the formula:

$$P(Y = j \mid X = x) = \frac{1}{\mathrm{K}} * \Sigma_{\mathrm{i}\in\mathrm{N}} 1_{y_i = j}$$

For a 2-class scenario, the KNN classifier predicts class j=1, if P(Y=j|X=x) > 0.5, and class j=0, otherwise.

In the cases of either classification, with classes 0 and 1, or where variables are on different scales, it may be the best to standardise the Hamming distances to between 0 and 1 as well. Further, the number of nearest neighbours is important, so k-folds cross-validation technique (see later) employing an independent dataset is used to determine the "optimal" (or "goldilocks") K-value. This K-value must have as low a bias and not overfit (i.e. for a low K-value) and as a low a variance and not too smooth (i.e. for a high K-value) as possible, which is commonly between K=3 and K=10. Between these K-values, the test error rate is commonly at a minimum.

Optimisation via k-folds cross-validation

When fitting a model using K-nearest neighbours (KNN), there is a trade-off between over-fitting a model to the training dataset (with poor fit to the test dataset) and having too simple a model that doesn't capture the variance in

the datasets. To assess this trade off, cross-validation is used, firstly by training a model on a large dataset (i.e. training set) and once an "optimal" solution is found, then this can be validated on the smaller test dataset (i.e. testing set). When this is done a method can be applied called k-folds cross-validation, which:

1. Cuts the training dataset into equally sized k-portions (say start with k=3)
2. Removes 1 portion to be used later
3. Tests the model on the remaining (k-1) portions
4. Validates this model on the removed portion
5. Repeats by removing each portion in turn until all portions are tested

This process is repeated for different k-values (i.e. 4, 5, etc.) up to a test limit to find the "optimal" k-value that gives the highest accuracy. This value will have the "optimal" trade-off between bias (i.e. for lower k-values) and variance (i.e. for higher k-values) and can be calculated from a plot of accuracy against number of neighbours.

Decision trees and random forests (bonus)

Both methods are supervised classification methods. The decision tree is designed with nodes / leaves of inputs, where a decision based on a Boolean argument, a greater than / less than, Yes-No, etc., results in 2 branches that represent a decision path. For example, a decision tree for eating ice cream above and below a temperature threshold of 20°C (Figure 4):

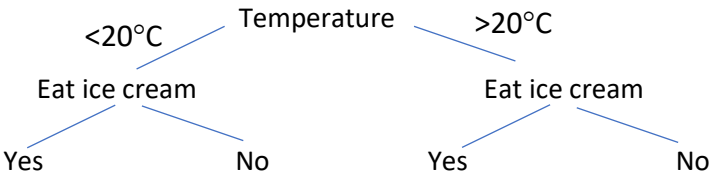

Figure 4: Decision tree diagram for eating ice cream related to temperature

This means that a decision tree makes a set of decisions from a number of parameters / inputs in the data, in this case temperature and whether people eat ice cream. Also, the reason we check the temperature parameter before asking the 'eat ice cream' parameter is because of the former's importance and the order that each input is used based on criteria in the Gini Index or from Information Gain.

However, many randomly generated decision trees may be used to make a final decision output more robust, and this is called a Random Forest. This process of combining decisions from different trees is an 'Ensemble Learning Process' where the final result takes the majority vote (or mean) of all the results from each decision tree.

Results of each classification method on the training and testing sets

The results provide the outputs from use of each classification method and the preferred optimisation technique, with comparison of the results given later. The results from the confusion matrices for decision tree and random forest analyses are also laid out as a bonus.

Linear Discriminant Analysis (LDA)

The LDA function is used to determine classes from analysis of the training set so that predictions can be made on the training set using the predict function. From LDA analysis on the testing set, this produces a set of prior probabilities and group means, as shown in tables 4a and 4b.

The posteriors from the predictions on the testing set are used to test against the known classes from the testing set. This is to calculate the True positive rate (TPR), the False positive rate (FPR) and the Area Under the Curve (AUC) using the performance function in the ROCR package. This package plots the ROC and gives an AUC value of = 0.761, which is an acceptable value (see Table 2), printed on the plot (Figure 5) as well as a table showing the "optimal" "cutoff" (p = 0.704) for the best combination of sensitivity and specificity (tables 5a and 5b).

	x
0	0.3225
1	0.6775

	X1	X2
0	0.2427896	-0.6143333
1	-0.1272712	0.3251450

Table 4: Results from LDA analysis on the training set with: (a) the prior probabilities; and (b) the group means

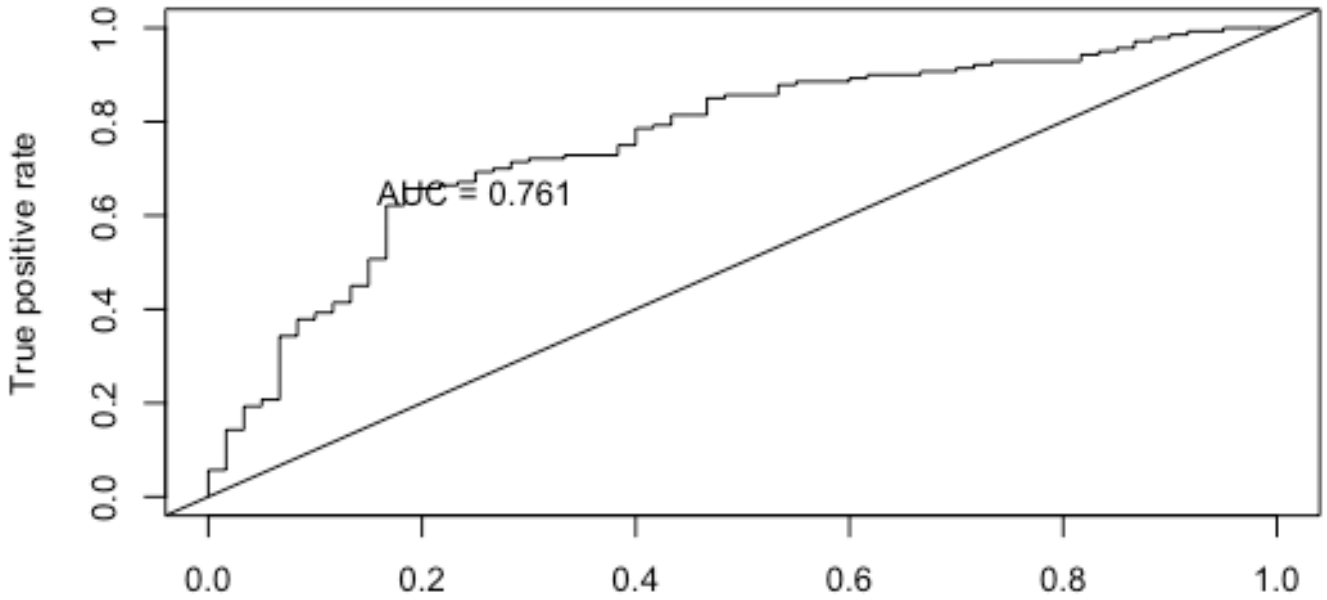

Figure 5: Receiver-Operator curve (ROC) for LDA analysis of X1-X2 data, showing an AUC value of 0.761 (acceptable)

sensitivity	0.6571429
specificity	0.8166667
cutoff	0.7041795

x	
accuracy	0.7550000
cutoff	0.5597111

Table 5: "Optimal" cutoffs (p) determined using the ROCR package of LDA analyses for the X1-X2 data of: (a) "optimal" sensitivity and specificity; as well as of: (b) "optimal" accuracy

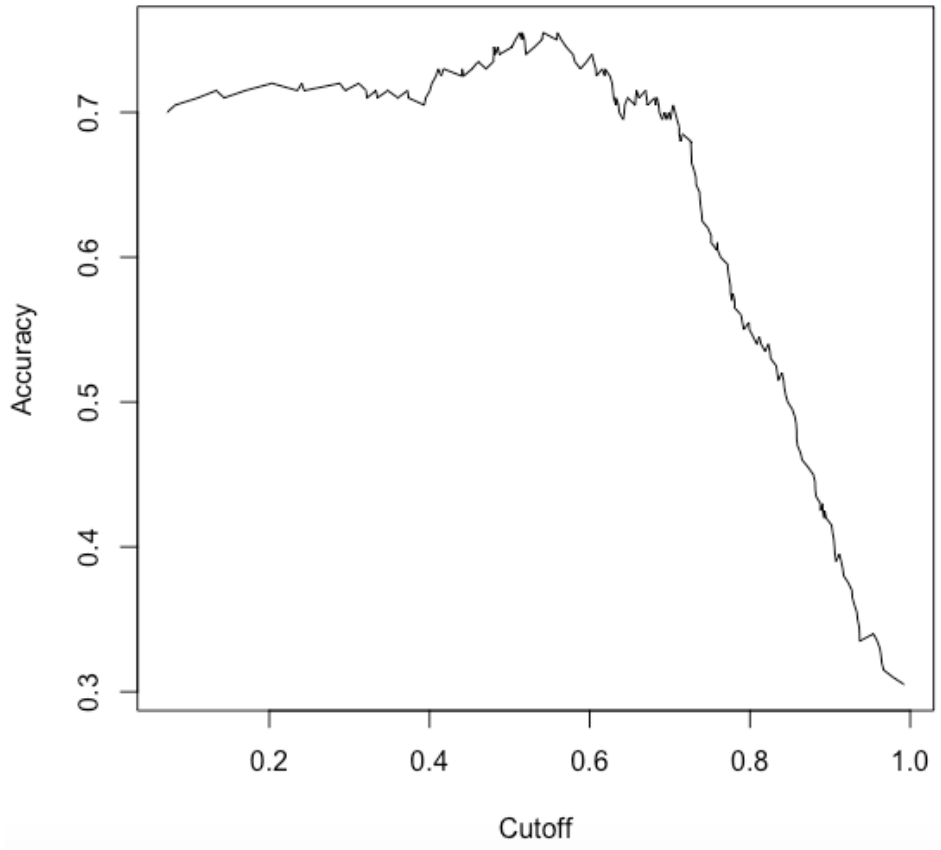

Figure 6: Line plot using LDA analyses in the ROCR package of accuracy against cutoff threshold (p) for the X1-X2 data

Further, the ROCR package plots the accuracy against the threshold cutoff (p) (Figure 6) in order to locate the "optimal" cutoff (p = 0.560) for the maximum accuracy (= 0.755), which is tabulated (tables 5a and 5b).

Quadratic Discriminant Analysis (QDA)

The QDA function is used to determine classes from analysis of the training set so that predictions can be made on the training set using the predict function. From QDA analysis on the testing set, this produces a set of prior probabilities and group means, as shown in tables 6a and 6b.

	x
0	0.3225
1	0.6775

	X1	X2
0	0.2427896	-0.6143333
1	-0.1272712	0.3251450

Table 6: Results from QDA analysis on the training set with: (a) the prior probabilities; and (b) the group means

The posteriors from the predictions on the testing set are used to test against the known classes from the testing set. This is to calculate the True positive rate (TPR), the False positive rate (FPR) and the Area Under the Curve (AUC) using the performance function in the ROCR package. This package plots the ROC and gives an AUC value of 0.895, which is an excellent value (see Table 2), printed on the plot (Figure 7) as well as a table showing the "optimal" cutoff (p = 0.635) for the best combination of sensitivity and specificity (tables 7a and 7b).

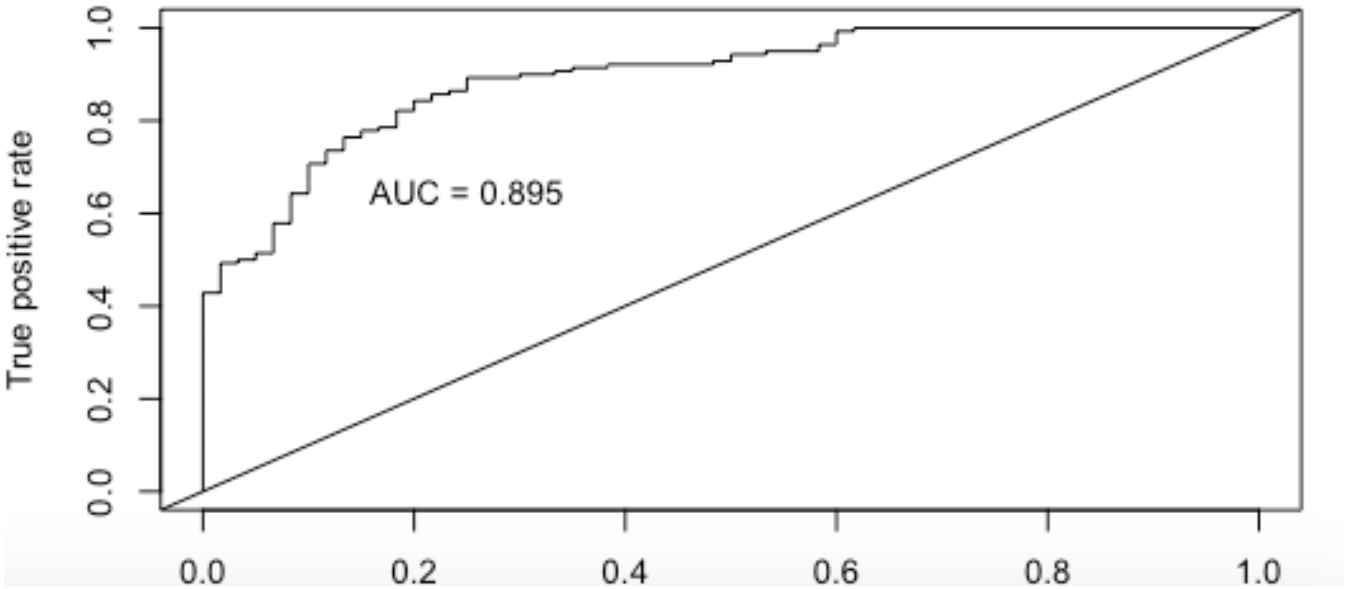

Figure 7: Receiver-Operator curve (ROC) for QDA analysis of X1-X2 data, showing an AUC value of 0.895 (excellent)

sensitivity	0.8428571
specificity	0.8000000
cutoff	0.6352704

	x
accuracy	0.8500000
cutoff	0.5924144

Table 7: "Optimal" cutoffs (p) determined using the ROCR package of QDA analyses for X1-X2 data of: (a) "optimal" sensitivity and specificity; as well as of: (b) "optimal" accuracy

Further, the ROCR package plots the accuracy against the cutoff (p) (Figure 8) in order to locate the "optimal" cutoff (p = 0.592) for the maximum accuracy (= 0.850), which is tabulated (tables 7a and 7b).

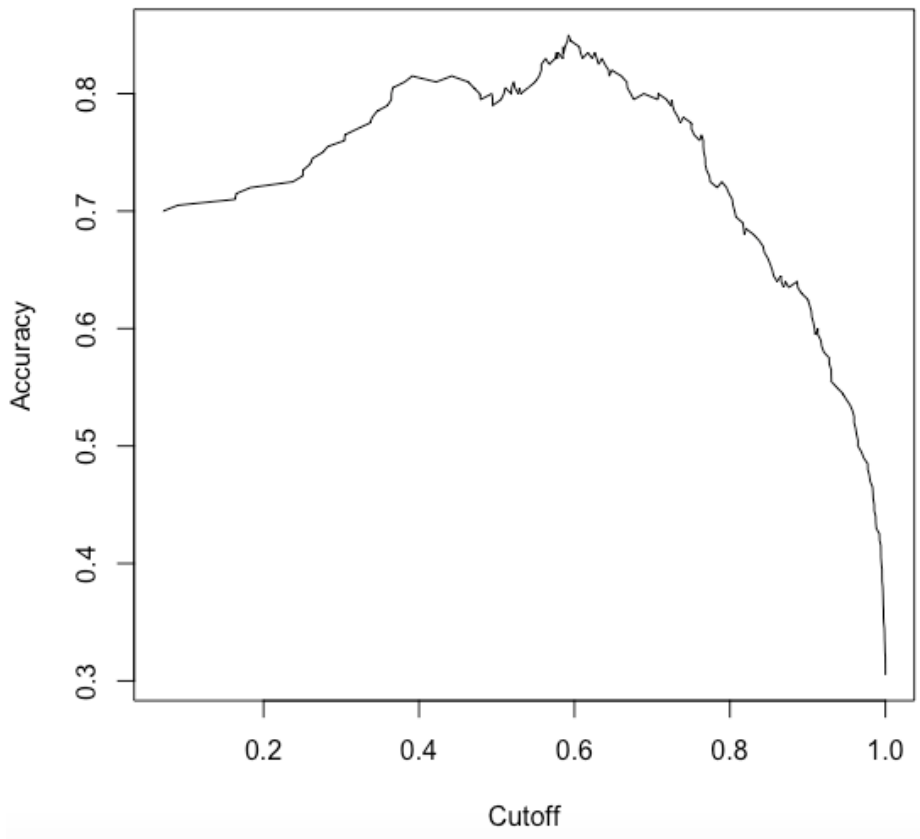

Figure 8: Line plot using QDA analyses in the ROCR package of accuracy against cutoff threshold (p) for X1-X2 data

Logistic regression

The logistic regression involves running a Generalised linear model (GLM) model comparing the variables, X1 and X2, with the Group using a binomial distribution on the training data. From the GLM model, the coefficients of X1 and X2 have significantly significant estimates with a negative slope for X1 and positive slope for X2 (Table 8a). This is used to determine classes from analysis of this training set, so that predictions can be made on the training set using the predict function. From logistic analysis on the testing set, this produces a confusion matrix of predicted values against observed values (Table 8b) as well as values for the accuracy, sensitivity and specificity (tables 9a and 9b). It should be noted that 55 predictions were incorrect from a possible 200 datapoints in the testing set, leading to an accuracy of 72.5%, specificity of ~82% but a sensitivity of only ~54%.

	x
(Intercept)	0.9391628
X1	-0.4275595
X2	1.1452598

	0	1
0	36	31
1	24	109

Table 8: Results from logistic analysis, showing: (a) estimates for the variables X1 (negative slope) and X2 (positive slope); and (b) predicted vs observed values matrix, with 55 predictions incorrect (from 200 datapoints)

	x
Accuracy	0.7250000

	x
Sensitivity	0.5373134
Specificity	0.8195489

Table 9: Results from the confusion matrix on the predicted vs. observed values from logistic analysis in Table (a), showing: (a) accuracy of the predictions (=0.725); and (b) sensitivity (=0.537) and specificity (=0.820) of the results

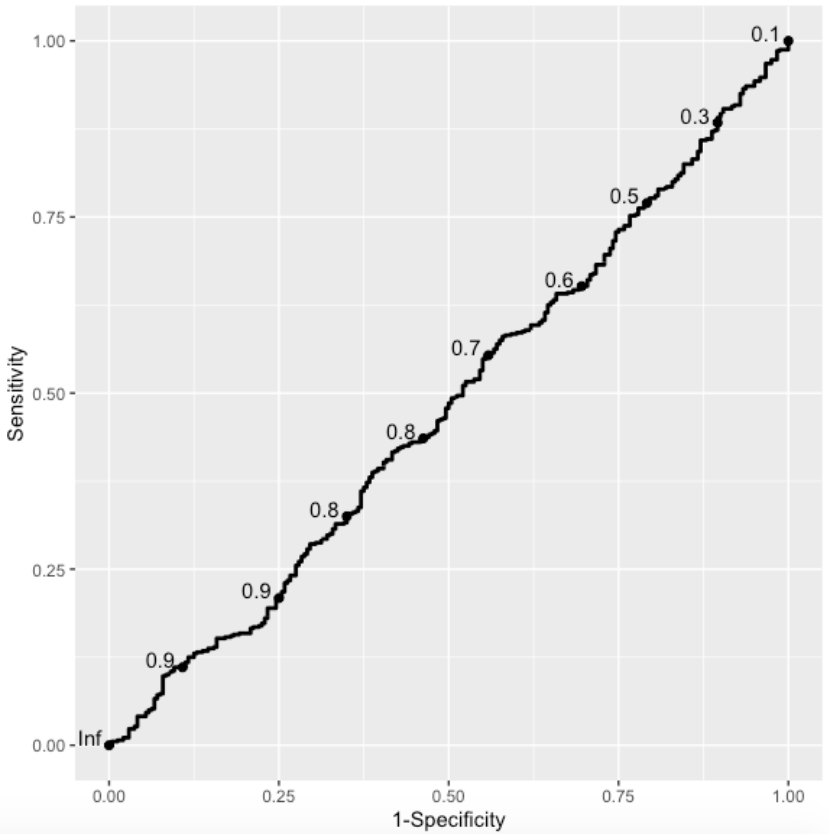

Figure 9: Receiver-Operator curve (ROC) for logistic analysis of X1-X2 data, showing an AUC value of 0.481 (poor)

The posteriors from the predictions on the testing set are used to test against the known classes from the testing set. This is to calculate the True positive rate (TPR), the False positive rate (FPR) and the Area Under the Curve (AUC) using the functions in the plotROC package combined with ggplot (Figure 9). This package plots the ROC and gives an AUC value of 0.481, which is a poor value (see Table 2) and also is a worse result than if the method had no discriminatory power on a dataset (AUC = 0.5). This suggests that this cross-validation technique is less effective at finding the accuracy of the model than with the confusion matrix.

Support Vector Machines

Support vector machines (SVM) are split into 3 types based on the type of hyperplane is used to separate the classes. The results of the Linear SVM (linear hyperplane), Radial SVM (circular hyperplane) and Poly SVM (polynomial hyperplane) are provided here:

Linear SVM

The Linear SVM function is used to determine classes from analysis of the training set so that predictions can be made on the training set using the predict function. From Linear SVM analysis on the testing set, this produces a confusion matrix of predicted values against observed values (Table 10b) as well as values for the accuracy, sensitivity and specificity (Table 10a). It should be noted that 51 predictions were incorrect from a possible 200 datapoints in the testing set, leading to an accuracy of 74.5%, specificity of ~89% but a sensitivity of only ~42%.

	x
Sensitivity	0.4166667
Specificity	0.8857143
Accuracy	0.7450000

	0	1
0	25	16
1	35	124

Table 10: Results from the confusion matrix showing: (a) accuracy of the predictions (=0.725), sensitivity (=0.537) and specificity (=0.820) of the results; and (b) the predicted vs. observed values from linear SVM analysis

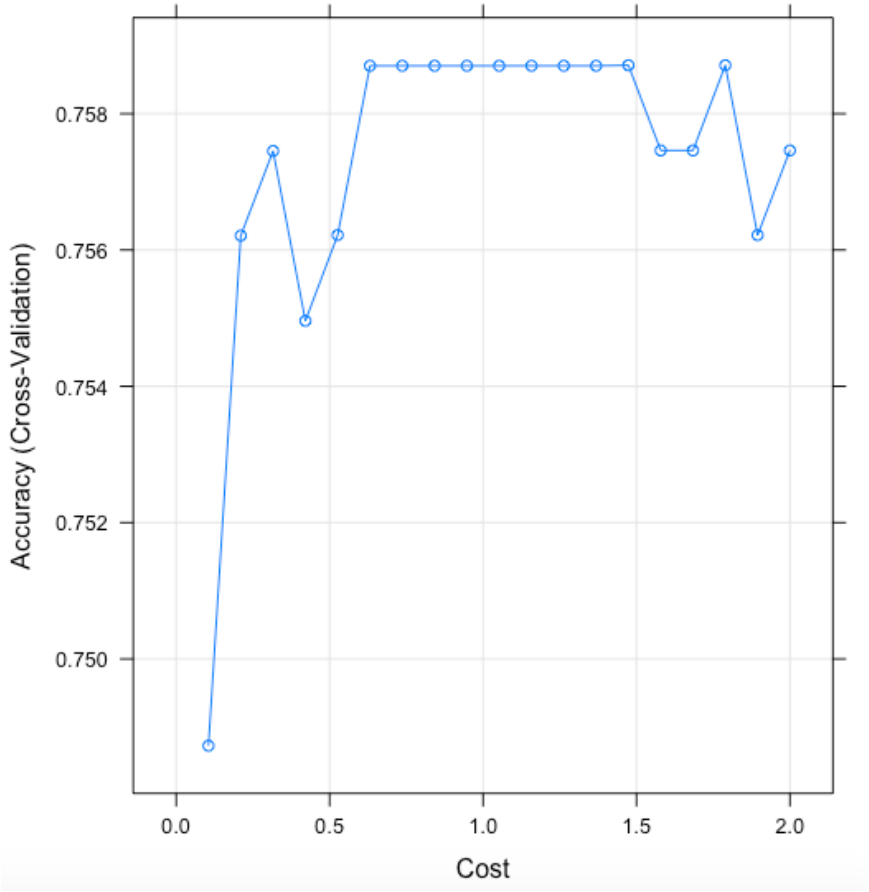

Figure 10: "Cost" against accuracy plot, showing results from the cross-validation of X1-X2 data using a Linear SVM

Using the linear SVM functions, the model is trained on the training set with cross-validation optimisation requiring "tuning" of the model output by providing a constraint between 0 and 2 for the "Cost" variable (C). This "Cost" parameter can be described as the weight of how many samples are inside the best fitting margin between the classes, with the best "Cost" value being ~1.474. This "tuning" means that the accuracy of the model can be maximised and is shown in a plot of "Cost" against accuracy (Figure 10), where maximum accuracy is ~76%. This is comparable to but slightly improved upon the result from the confusion matrix.

Radial SVM

The Radial SVM function is used to determine classes from analysis of the training set so that predictions can be made on the training set using the predict function. From Radial SVM analysis on the testing set, this produces a confusion matrix of predicted values against observed values (Table 11b) as well as values for the accuracy, sensitivity and specificity (Table 11a). It should be noted that 37 predictions were incorrect from a possible 200 datapoints in the testing set, leading to an accuracy of 81.5%, specificity of ~92% but a sensitivity of only ~57%.

	x
Sensitivity	0.5666667
Specificity	0.9214286
Accuracy	0.8150000

	0	1
0	34	11
1	26	129

Table 11: Results from the confusion matrix showing: (a) accuracy of the predictions (=0.815), sensitivity (=0.567) and specificity (=0.921) of the results; and (b) the predicted vs. observed values from radial SVM analysis

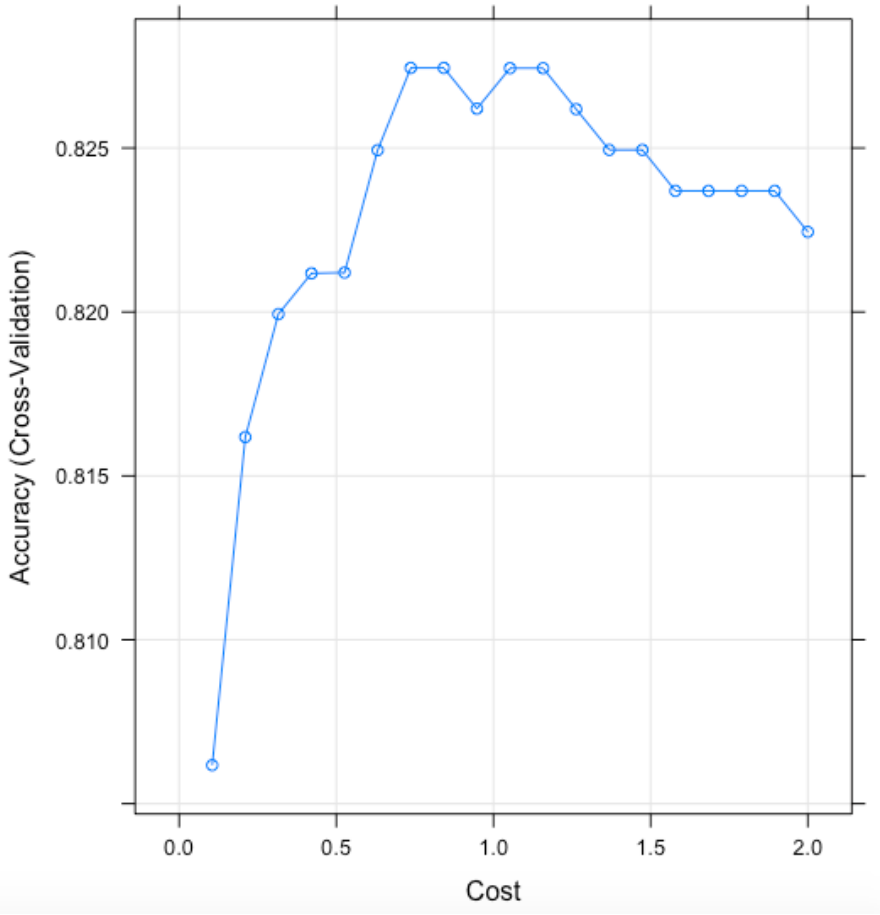

Figure 11: "Cost" against accuracy plot, showing results from the cross-validation of X1-X2 data using a Radial SVM

Using the radial SVM functions, the model is trained on the training set with cross-validation optimisation requiring "tuning" of the model output by providing a constraint between 0 and 2 for the "Cost" variable (C). This "Cost" parameter can be described as the weight of how many samples are inside the best fitting margin between the classes, with the best "Cost" value being ~1.600. This "tuning" means that the accuracy of the model can be maximised and is shown in a plot of "Cost" against accuracy (Figure 11), where maximum accuracy is ~82.5%. This is comparable to but slightly improved upon the result from the confusion matrix.

Polynomial SVM

The Polynomial SVM function is used to determine classes from analysis of the training set so that predictions can be made on the training set using the predict function. From Polynomial SVM analysis on the testing set, this produces a confusion matrix of predicted values against observed values (Table 12b) as well as values for the accuracy, sensitivity and specificity (Table 12a). It should be noted that 36 predictions were incorrect from a possible 200 datapoints in the testing set, leading to an accuracy of 82%, specificity of ~94% but a sensitivity of only ~55%.

	x
Sensitivity	0.5500000
Specificity	0.9357143
Accuracy	0.8200000

	0	1
0	33	9
1	27	131

Table 12: Results from the confusion matrix showing: (a) accuracy of the predictions (=0.820), sensitivity (=0.550) and specificity (=0.936) of the results; and (b) the predicted vs. observed values from polynomial SVM analysis

Using the polynomial SVM functions, the model is trained on the training set with cross-validation optimisation requiring "tuning" of the model output by providing a constraint between 0 and 2 for the "Cost" variable (C). This

"Cost" parameter can be described as the weight of how many samples are inside the best fitting margin between the classes, with the best "Cost" value being ~1.600. This "tuning" means that the accuracy of the model can be maximised and is shown in a plot of "Cost" against accuracy (Figure 12), where maximum accuracy is ~81.5%. This is comparable to but slightly reduced from the result in the confusion matrix.

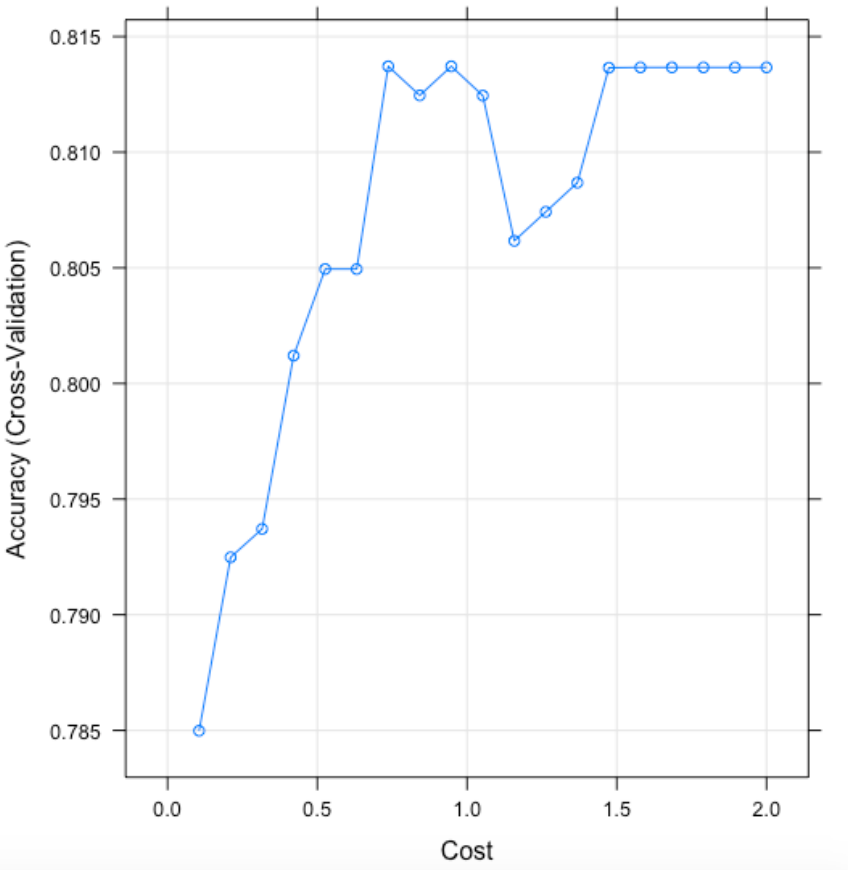

Figure 12: "Cost" against accuracy plot, showing results from cross-validation of X1-X2 data using a polynomial SVM

K-nearest neighbours

With k-nearest neighbours, the trainControl function is used to determine the "optimal" number of k using the training dataset. This process undertakes a 10-fold cross-validation that is repeated 5 times and gave an "optimal" k-value of 13. This result is shown in the plot of accuracy against number of neighbours (Figure 13), which gives a maximum accuracy of ~0.82.

This k-value is used to make predictions on the training set using the predict function. From this analysis on the testing set, this produces a confusion matrix of predicted values against observed values (Table 13b) as well as values for the accuracy, sensitivity and specificity (Table 13a). It should be noted that 36 predictions were incorrect from a possible 200 datapoints in the testing set, leading to an accuracy of 82%, specificity of ~91% but a sensitivity of ~62%. This accuracy figure is comparable to but slightly improved upon the result from the confusion matrix.

	x
Sensitivity	0.6166667
Specificity	0.9071429
Accuracy	0.8200000

	0	1
0	37	13
1	23	127

Table 13: Results from the confusion matrix showing: (a) accuracy of the predictions (=0.820), sensitivity (=0.617) and specificity (=0.907) of the results; and (b) the predicted vs. observed values from polynomial SVM analysis

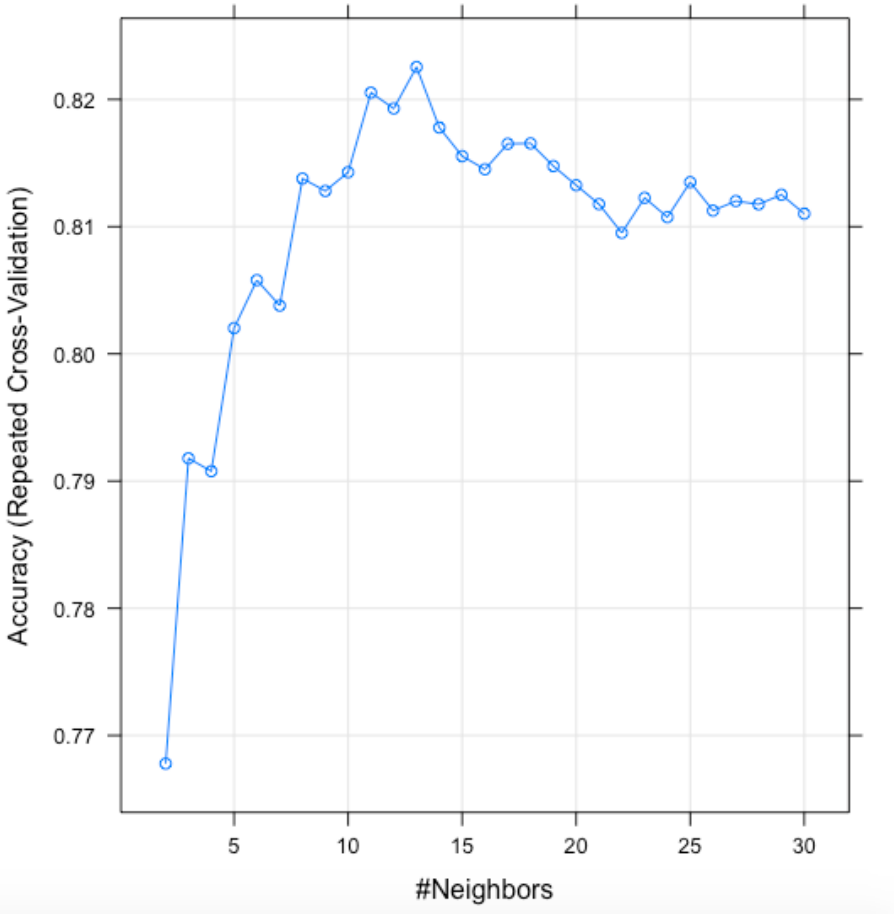

Figure 13: Number of neighbours against accuracy plot, showing results of the X1-X2 data analysis using the k-folds cross-validation technique

Bonus results on decision trees and random forests

Decision tree and random forest analyses where undertaken on the classification dataset as bonus analyses and have been optimised by use of confusion matrices, with the results shown here.

Decision trees

A decision tree was used to determine classes from analysis of the training set so that predictions can be made on the training set using the predict function. From decision tree analysis on the testing set, this produces a confusion matrix of predicted values against observed values (Table 14b) as well as values for the accuracy, sensitivity and specificity (Table 14a). It should be noted that 44 predictions were incorrect from a possible 200 datapoints in the testing set, leading to an accuracy of 78%, specificity of ~93% but a sensitivity of only ~43%.

	x
Sensitivity	0.4333333
Specificity	0.9285714
Accuracy	0.7800000

	0	1
0	26	10
1	34	130

Table 14: Results from the confusion matrix showing: (a) accuracy of the predictions (=0.780), sensitivity (=0.433) and specificity (=0.929) of the results; and (b) the predicted vs. observed values from decision tree analysis

Random forests

A random forest was used to determine classes from analysis of the training set so that predictions can be made on the training set using the predict function. From random forest analysis on the testing set, this produces a confusion matrix of predicted values against observed values (Table 15b) as well as values for the accuracy, sensitivity and

specificity (Table 15a). It should be noted that 39 predictions were incorrect from a possible 200 datapoints in the testing set, leading to an accuracy of 80.5%, specificity of ~86% and a sensitivity of ~68%.

	x
Sensitivity	0.6833333
Specificity	0.8571429
Accuracy	0.8050000

	0	1
0	41	20
1	19	120

Table 15: Results from the confusion matrix showing: (a) accuracy of the predictions (=0.805), sensitivity (=0.683) and specificity (=0.857) of the results; and (b) the predicted vs. observed values from random forest analysis

Discussion and Conclusions on comparison of results

Comparison using the different classification methods

The different methods require different analyses, produce different outputs and use different optimisation methods. However, accuracy, sensitivity and specificity results from all the methods can be extracted and so can be compared.

For the two discriminant analyses and logistic regression, QDA produces a ROC curve with the highest AUC value (=0.895) and at the relative "cutoff" values (p) for each analysis produces a higher accuracy (=85%) and higher sensitivity (~84%) without losing too much specificity (=80%).

For the support vector machines and k-nearest neighbours that used analyses both from confusion matrices and cross-validation techniques (i.e. “Cost” vs. accuracy and k-folds, respectively), the highest accuracy and specificity are from a Support Vector Machine using a polynomial method (=82% and ~93.6%, respectively), whilst the highest sensitivity is from a k-nearest neighbour method (=61.67%). Overall, the k-nearest neighbours model loses the least with respect to accuracy and specificity (=82% and ~92.1%, respectively), whilst retaining a sensitivity of 61.67%.

Therefore, the best method that retains the highest accuracy, sensitivity and specificity is QDA, so is the preferred method. These methods have also been tested on a dataset (classification.True.csv) that contains the same data, but without the random noise component added to test whether QDA remains as the best overall method.

Comparison of results from this analysis with the true classification data

The results from the data in Classification.csv and ClassificationTrue.csv have been compared and the comparative results will be discussed method by method, with a final overall summary:

- LDA – The "True" dataset displays improved accuracy (=78%) and sensitivity (=74%), with a minor reduction in specificity (=78%) compared with the initial dataset. There is an improvement in AUC in the ROC curve to 0.816.
- QDA – The "True" dataset displays improved accuracy (=89.5%) and specificity (=90%), with the same sensitivity (=84%) compared with the initial dataset. There is an improvement in AUC in the ROC curve to 0.945.
- Logistic regression (glm) – The "True" dataset displays improved accuracy (=75%) and specificity (=53%), with a slight reduction in sensitivity (=50%) compared with the initial dataset.
- Logistic regression (confusion matrix) - The "True" dataset displays improved accuracy (=76.5%), specificity (~83.7%), with a slight reduction in sensitivity (~53.2%) compared with the initial dataset.
- SVM with linear kernel – The "True" dataset displays improved accuracy (=76.5%), specificity (~85.3%), and also in sensitivity (50%) compared with the initial dataset.
- SVM with radial kernel – The "True" dataset displays improved accuracy (=87.5%), specificity (~92.7%), and also in sensitivity (72%) compared with the initial dataset.
- SVM with polynomial kernel – The "True" dataset displays improved accuracy (=86.5%), sensitivity (=68%), and a slight reduction in specificity (~92.7%) compared with the initial dataset.
- K-nearest neighbours – The "True" dataset displays improved accuracy (=87.5%), sensitivity (=72%), and also in specificity (~92.7%) compared with the initial dataset.

In all cases, there was an improvement in the accuracy and in at least one of the sensitivity or specificity for the "True" dataset. However, with the improvements, the best and preferred model to use continues to be QDA.

The bonus methods can also be compared with the results above, with the comparison of outcomes for decision trees and random forests given here:

- Decision Tree - The "True" dataset displays improved accuracy (=88.5%), specificity (~95.3%), and also in sensitivity (68%) compared with the initial dataset.
- Random Forest - The "True" dataset displays improved accuracy (=89%), specificity (~93.3%), and also in sensitivity (76%) compared with the initial dataset.

From these two extra analyses, it can be seen that random forests analysis is the method with higher accuracy, sensitivity and specificity of the two methods. When compared to the other methods, QDA is still slightly better on accuracy, sensitivity and specificity than random forests, so remains as the preferred method.

Appendix – Code for Classification

```
## B. Classification [48 marks]

# The following figure shows the information in the dataset Classification.csv - it shows two different
# groups, plotted against two explanatory variables. This is simulated data - the groupings are determined
# by a (known, but not to you!) function of X1 and X2 with added noise/random error. The aim is to find a
# suitable method for classifying the 1000 datapoints into the two groups from a selection of possible approaches.

# Load required packages for classification
library(dplyr)
library(caret)
library(class) # this has the knn function
library(e1071)
library(MASS)
library(tidyverse)
library(readr)
library(repr)
library(glmnet)
library(ROCR)
library(C50)
library(kernlab)
library(randomForest)
library(plotROC) # has geom_roc
library(kableExtra)


## 1. [3 marks] Summarise the two groups in terms of the variables X1 and X2. Describe your findings.

# Read in the Classification.csv file and check the header data
class <- read.csv("Classification.csv")
head(class)

# Look at the data
group.class <- class[,3]
X1.class <- class[,1]
X2.class <- class[,2]

# Basic Scatterplot Matrix of X1 and X2
pairs(~ group.class + X1.class + X2.class, data = class,
    main = "Simple Scatterplot Matrix of X1 and X2")

# ggplot of X1-X2 data
ggplot(data = class, aes(x = X1.class, y = X2.class, color = group.class)) +
  geom_point(size = 2)

# Summary statistics of class dataset
summary(class)
kbl(summary(class)) %>% kable_styling()
#       X1                X2              Group
# Min.   :-3.055858   Min.   :-3.41429   Min.   :0.000
# 1st Qu.:-0.679636   1st Qu.:-0.62566   1st Qu.:0.000
# Median :-0.038071   Median : 0.01472   Median :1.000
# Mean   : 0.002033   Mean   : 0.01631   Mean   :0.682
# 3rd Qu.: 0.649037   3rd Qu.: 0.66946   3rd Qu.:1.000
# Max.   : 3.285469   Max.   : 3.94396   Max.   :1.000
```

```
# Comparing group.class with X1.class shows that there is a narrow range between X1.class values of -1 to 1
# for group.class == 0 and wide range between X1.class values of -2 to 2 for group.class == 1
# Comparing group.class with X2.class shows that there is a wide range between X2.class values of -2 to 2
# for group.class == 0 and wide range between X2.class values of -1 to 3 for group.class == 1
# Comparing X1.class with X2.class shows well scattered data points between -3 and 3 for both parameters
# IQR for both X1.class and X2.class are approximately between -0.67 and 0.67, with both the mean and median ~ 0



## 2. [2 marks] Select 80% of the data to act as a training set, with the remaining 20% for testing/evaluation.

# Next we manually define a function that will split the data into a training set and a test set,
# and use this function to make the split:

# Define function that will split the data into training and test sets
trainTestSplit <- function(df.class, trainPercent, seed2){
  ## Sample size percent
  smp.size.class <- floor(trainPercent/100 * nrow(df.class))
  ## set the seed
  set.seed(seed2)
  train1 <- sample(seq_len(nrow(df.class)), size = smp.size.class)
  train1
}

# Split as training and test sets
train1 <- trainTestSplit(class, trainPercent = 80, seed = 321)
train.class <- as.data.frame(class[train1, ])
test.class <- as.data.frame(class[-train1, ])

count(class)
# 1000
length(train1)
# 800
count(train.class)
# 800
count(test.class)
# 200



## 3. Perform classification using the following methods. In each case, briefly describe how the method works,
## present the results of an evaluation of the method and describe your findings. Where appropriate optimise
## the parameters of the method (e.g. by using ROC curve, cross validation). In each case describe carefully
## how the optimisation method works.



## (a) [5 marks] Linear discriminant analysis.

## Now that our data is ready, we can use the lda() function in R to do our analysis,
## which is functionally identical to the lm() and glm() functions:
f.lda.class <- paste(names(train.class)[3], "~", paste(names(train.class)[-3], collapse = " + "))
class.lda <- lda(as.formula(paste(f.lda.class)), data = train.class)

## Now let's make some predictions on our testing-data:
class.lda.predict <- predict(class.lda, newdata = test.class)

## we can use the lda() function in R to perform our analysis.
```

```
lda.result <- lda(as.factor(Group) ~ X1 + X2, data = train.class)
kbl(lda.result$prior) %>% kable_styling()
kbl(lda.result$means) %>% kable_styling()
##
## Prior probabilities of groups:
##    0    1
## 0.3225 0.6775
##
## Group means:
##        X1         X2
## 0  0.2427896 -0.6143333
## 1 -0.1272712  0.3251450
##
## Coefficients of linear discriminants:
##        LD1
## X1 -0.3860421
## X2  1.0170460

## Finally we can look at the predictions and compare them to the test/evaluation dataset just
## as we did for other methods.

## Evaluation and construction of ROC AUC plot:
## Get the posteriors as a dataframe.
class.lda.predict.posteriors <- as.data.frame(class.lda.predict$posterior)
kbl(class.lda.predict.posteriors) %>% kable_styling()
## Evaluate the model
pred.lda.class <- prediction(class.lda.predict.posteriors[,2], as.factor(test.class[,3]))
roc.perf.lda.class <- performance(pred.lda.class, measure = "tpr", x.measure = "fpr")
auc.train.lda.class <- performance(pred.lda.class, measure = "auc")
auc.train.lda.class <- auc.train.lda.class@y.values
## Plot
plot(roc.perf.lda.class)
abline(a = 0, b = 1)
text(x = 0.25, y = 0.65, paste("AUC = ", round(auc.train.lda.class[[1]], 3), sep = ""))

## To find the optimal cutoff point
opt.cut.lda <- function(roc.perf.lda.class, pred.lda.class){
  cut.ind.lda <- mapply(FUN = function(x, y, p){
    d = (x - 0)^2 + (y-1)^2
    ind = which(d == min(d))
    c(sensitivity = y[[ind]], specificity = 1-x[[ind]],
      cutoff = p[[ind]])
  }, roc.perf.lda.class@x.values, roc.perf.lda.class@y.values, pred.lda.class@cutoffs)
}

## Print the sensitivity, specificity and cutoff value at the "optimal" value
print(opt.cut.lda(roc.perf.lda.class, pred.lda.class))
kbl(opt.cut.lda(roc.perf.lda.class, pred.lda.class)) %>% kable_styling()
## sensitivity 0.6571429
## specificity 0.8166667
## cutoff(p)   0.5597111

## The “optimal” point (i.e. point that appears to be closest to the top left-hand corner)
## has a TRP of ~65%, a FPR of ~18% and a cutoff of ~0.70. The AUC is 0.761, which is acceptable.
## This means that 18% of the points that we believe are correct are in the wrong class
## and 65% of points that we believe are correct are actually in the correct class.
```

```
## Also, let's get the overall accuracy for the lda predictions and plot it:
acc.lda.perf <- performance(pred.lda.class, measure = "acc")
plot(acc.lda.perf)

## And find the maximum accuracy at the maximum cutoff
ind.lda <- which.max(slot(acc.lda.perf, "y.values")[[1]])
lda.acc <- slot(acc.lda.perf, "y.values")[[1]][ind.lda]
cutoff.lda <- slot(acc.lda.perf, "x.values")[[1]][ind.lda]
print(c(accuracy = lda.acc, cutoff = cutoff.lda))
kbl(c(accuracy = lda.acc, cutoff = cutoff.lda)) %>% kable_styling()
## accuracy  cutoff(p)
## 0.7550000 0.5597111

## The accuracy value from the lda accuracy plot at a cutoff (p) of 0.8 is approximately 55%
## but could be as high as 75.5% at a cutoff(p) of approximately 0.56.



## (b) [5 marks] Quadratic discriminant analysis.

## Now that our data is ready, we can use the qda() function in R to do our analysis:
f.qda.class <- paste(names(train.class)[3], "~", paste(names(train.class)[-3], collapse = " + "))
class.qda <- qda(as.formula(paste(f.qda.class)), data = train.class)

## Now let's make some predictions on our testing-data:
class.qda.predict <- predict(class.qda, newdata = test.class)

## we can use the qda() function in R to perform our analysis.
qda.result <- qda(as.factor(Group) ~ X1 + X2, data = train.class)
kbl(qda.result$prior) %>% kable_styling()
kbl(qda.result$means) %>% kable_styling()
##
## Prior probabilities of groups:
##    0    1
## 0.3225 0.6775
##
## Group means:
##        X1         X2
## 0  0.2427896 -0.6143333
## 1 -0.1272712  0.3251450
##

## Finally we can look at the predictions and compare them to the test/evaluation dataset just
## as we did for other methods.

## Evaluation and construction of ROC AUC plot:
## Get the posteriors as a dataframe.
class.qda.predict.posteriors <- as.data.frame(class.qda.predict$posterior)
kbl(class.qda.predict.posteriors) %>% kable_styling()
## Evaluate the model
pred.qda.class <- prediction(class.qda.predict.posteriors[,2], as.factor(test.class[,3]))
roc.perf.qda.class <- performance(pred.qda.class, measure = "tpr", x.measure = "fpr")
auc.train.qda.class <- performance(pred.qda.class, measure = "auc")
auc.train.qda.class <- auc.train.qda.class@y.values
## Plot
plot(roc.perf.qda.class)
abline(a = 0, b = 1)
```

```
text(x = 0.25, y = 0.65, paste("AUC = ", round(auc.train.qda.class[[1]], 3), sep = ""))

## To find the optimal cutoff point
opt.cut.qda <- function(roc.perf.qda.class, pred.qda.class){
  cut.ind.qda <- mapply(FUN = function(x, y, p){
    d = (x - 0)^2 + (y-1)^2
    ind = which(d == min(d))
    c(sensitivity = y[[ind]], specificity = 1-x[[ind]],
      cutoff = p[[ind]])
  }, roc.perf.qda.class@x.values, roc.perf.qda.class@y.values, pred.qda.class@cutoffs)
}

## Print the sensitivity, specificity and cutoff value at the "optimal" value
print(opt.cut.qda(roc.perf.qda.class, pred.qda.class))
kbl(opt.cut.qda(roc.perf.qda.class, pred.qda.class)) %>% kable_styling()
## sensitivity 0.8428571
## specificity 0.8000000
## cutoff      0.5924144

## The “optimal” point (i.e. point that appears to be closest to the top left-hand corner)
## has a TRP of ~84%, a FPR of 20% and a cutoff of ~0.59. The AUC is 0.895, which is excellent.
## This means that 20% of the points that we believe are correct are in the wrong class
## and 84% of points that we believe are correct are actually in the correct class.
## The AUC value with qda is great improvement on lda.

## Finally, let's get the overall accuracy for the qda predictions and plot it:
acc.qda.perf = performance(pred.qda.class, measure = "acc")
plot(acc.qda.perf)

## And find the maximum accuracy at the maximum cutoff
ind.qda <- which.max(slot(acc.qda.perf, "y.values")[[1]])
qda.acc <- slot(acc.qda.perf, "y.values")[[1]][ind.qda]
cutoff.qda <- slot(acc.qda.perf, "x.values")[[1]][ind.qda]
print(c(accuracy = qda.acc, cutoff = cutoff.qda))
kbl(c(accuracy = qda.acc, cutoff = cutoff.qda)) %>% kable_styling()
## accuracy  cutoff(p)
## 0.8500000 0.5924144

## The accuracy value from the lda accuracy plot at a cutoff (p) of 0.8 is approximately 70%
## but could be as high as 85% at a cutoff(p) of approximately 0.59.



## (c) [8 marks] Logistic regression.

## Next let’s start by fitting a logistic regression will all the variables using the
## glm() function, with family = 'binomial'.

## Here we fit a logistic model to the data
logit.class <- glm(as.factor(Group) ~ X1 + X2, family = "binomial", data = train.class)

# Plot logistic model residuals vs. fitted, normal qq, scale-location and residual vs. leverage plots
par(mar = c(2,4,4,2), cex = 1.0)
layout(matrix(c(1,1,2,2),2,2,byrow=TRUE))
plot(logit.class)

# Look at the summary of the logistic model
```

```
summary(logit.class)
logit.class$coefficients
kbl(logit.class$coefficients) %>% kable_styling()
##
## Deviance Residuals:
##    Min      1Q      Median      3Q     Max
##  -2.4281  -0.9154   0.4811   0.7732   2.1791
##
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept)  0.93916    0.09099  10.321  < 2e-16 ***
## X1          -0.42756    0.09150  -4.673 2.97e-06 ***
## X2           1.14526    0.10475  10.934  < 2e-16 ***
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 1005.98 on 799  degrees of freedom
## Residual deviance:  816.84 on 797  degrees of freedom
## AIC: 822.84
##
## Number of Fisher Scoring iterations: 4

# Run the model on the test set
test.class.probs <- predict(logit.class, newdata = test.class[,1:2])
pred.logit.class <- rep(0, length(test.class.probs))
pred.logit.class[test.class.probs >= 0.5] <- 1

# Create table to compare test set and predictions
table(pred.logit.class, test.class$Group)
kbl(table(pred.logit.class, test.class$Group)) %>% kable_styling()
##
## pred.logit.class   0   1
##                0  36  31
##                1  24 109

# Set up ConfusionMatrix to compare test set and predictions from the logistic model
logit.conmat <- confusionMatrix(as.factor(test.class[,3]), as.factor(pred.logit.class))
kbl(logit.conmat$overall) %>% kable_styling()
kbl(logit.conmat$byClass) %>% kable_styling()
## Confusion Matrix and Statistics
##
##          Reference
## Prediction   0   1
##          0  36  24
##          1  31 109

## Accuracy : 0.725
## 95% CI : (0.6576, 0.7856)
## No Information Rate : 0.665
## P-Value [Acc > NIR] : 0.0408

## Kappa : 0.3664
## Mcnemar's Test P-Value : 0.4185

##             Sensitivity : 0.5373
##             Specificity : 0.8195
##          Pos Pred Value : 0.6000
```

```
##         Neg Pred Value : 0.7786
##             Prevalence : 0.3350
##         Detection Rate : 0.1800
##   Detection Prevalence : 0.3000
##      Balanced Accuracy : 0.6784
##       'Positive' Class : 0
##
```

This gets 55 points incorrect (out of 200), so has an accuracy of 72.5%, with a sensitivity of only 53.7%
and specificity of ~82%. This is good but the caret package will be used create a machine-learned logistic
model to attempt to improve on this accuracy.

```
# Use the fitted model to get the probability that a given obs belongs to class 1 using the predict function.
class.logit.predict <- predict(logit.class, type = "response")

# In order to use the geom_roc function we have to create a dataframe that contains the estimated
# probabilities and the actual class labels.

# Create a dataframe that has the true labels, and the probabilities that the glm fit gives for
# belonging to class 1 by simulating some data
df.logit.class <- data.frame(a = test.class[,3], b = predict(logit.class, type = "response"))

# Use ggplot with the geom_roc option to get the ROC curve. The asthetics of geom_roc are
# the true class labels, d, and the associated probabilities of belonging to class 1, m.
ROC.logit.class <- ggplot(df.logit.class, aes(d = a, m = b)) +
  geom_roc() + ylab("Sensitivity") + xlab("1-Specificity")
ROC.logit.class

# Finally, we can get the area under the receiver operating curve by using the calc_auc function
# on the previous ggplot object - It's actually worse than random chance (AUC < 0.5).
calc_auc(ROC.logit.class)$AUC
# [1] 0.4805878
```

Comparing the two optimisation methods, there appears to be a contrast in results based on using a Confusion Matrix,
which gives an accuracy of 72.5%, a sensitivity of 53.7% and a specificity of ~82%, and using a ROC curve,
with an AUC value of 0.48 (suggesting a worse result than random chance). Consequently, the result from the
Confusion Matrix is the more reliable (i.e. "less confused") method and is preferred here.

```
## Now use the caret package to train the machine to fit a logistic model
modelFit.logit.class <- train(as.factor(Group) ~ X1 + X2, data = train.class,
                      method = "glm", family = binomial(link = "logit"),
                      preProcess = c('scale', 'center'))

# Look at the summary of the machine-trained model
logit.summary <- summary(modelFit.logit.class)
kbl(table(pred.logit.class, test.class$Group)) %>% kable_styling()
##
## Call:
## NULL
##
## Deviance Residuals:
##    Min      1Q      Median      3Q     Max
##  -2.4281  -0.9154   0.4811   0.7732   2.1791
```

```
##
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept)  0.93916    0.09099  10.321  < 2e-16 ***
## X1          -0.42756    0.09150  -4.673 2.97e-06 ***
## X2           1.14526    0.10475  10.934  < 2e-16 ***
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 1005.98 on 799  degrees of freedom
## Residual deviance: 816.84  on 797  degrees of freedom
## AIC: 822.84
##
## Number of Fisher Scoring iterations: 4


## Create confusionMatrix to compare test data and prediction from the machine-trained model
confusionMatrix(as.factor(test.class[,3]), as.factor(predict(modelFit.logit.class, test.class)))
##
##          Reference
## Prediction   0   1
##          0  25  35
##          1  16 124

## Accuracy : 0.745
## 95% CI : (0.6787, 0.8039)
## No Information Rate : 0.795
## P-Value [Acc > NIR] : 0.96433

## Kappa : 0.3325
## Mcnemar's Test P-Value : 0.01172

##             Sensitivity : 0.6098
##             Specificity : 0.7799
##          Pos Pred Value : 0.4167
##          Neg Pred Value : 0.8857
##              Prevalence : 0.2050
##          Detection Rate : 0.1250
##    Detection Prevalence : 0.3000
##       Balanced Accuracy : 0.6948
##       'Positive' Class : 0

## The machine-trained model obtains about 74.5% accuracy overall (51 of 200 points were incorrect),
## with a sensitivity of 78% and specificity of only 41.67%. This is still good and is also an
## improvement on the 72% prediction accuracy from the initial logistic model. This suggests that
## a machine can be trained to create a model with better accuracy.


# Use the fitted model to get the probability that a given obs belongs to class 1 using the predict function.
predict(modelFit.logit.class, type = "raw")

# In order to use the geom_roc function we have to create a dataframe that contains the estimated
# probabilities and the actual class labels.

# Create a dataframe that has the true labels, and the probabilities that the glm fit gives for
# belonging to class 1 by simulating some data
df.modelFit.logit.class <- data.frame(a = test.class[,3], b = predict(modelFit.logit.class, type = "raw"))
```

```
# Use ggplot with the geom_roc option to get the ROC curve. The asthetics of geom_roc are
# the true class labels, d, and the associated probabilities of belonging to class 1, m.
ROC.modelFit.logit.class <- ggplot(df.modelFit.logit.class, aes(d = a, m = b)) +
  geom_roc() + ylab("Sensitivity") + xlab("1-Specificity")
ROC.modelFit.logit.class

# Finally, we can get the area under the receiver operating curve by using the calc_auc function
# on the previous ggplot object - It's actually worse than random chance (AUC < 0.5).
calc_auc(ROC.modelFit.logit.class)$AUC
# [1] 0.5



## (d) [10 marks] Support vector machines.

## SVM with linear kernel ##

set.seed(1066)
mdl.linearSVM.class <- train(x = train.class[,1:2], y = as.factor(train.class[,3]), method = "svmLinear")
print(mdl.linearSVM.class)
# 800 samples
# 2 predictors
# 2 classes: '0', '1'

# No pre-processing
# Resampling: Bootstrapped (25 reps)
# Summary of sample sizes: 800, 800, 800, 800, 800, 800, ...
# Resampling results:

#  Accuracy   Kappa
# 0.7381556  0.3501186

# Tuning parameter 'C' was held constant at a value of 1


# Test model on testing data (yTestPred <- mdl %>% predict(xTest))
testPred.linearSVM.class <- predict(mdl.linearSVM.class, newdata = test.class[,1:2])
# predicted vs. true
linearSVM.conmat <- confusionMatrix(as.factor(testPred.linearSVM.class), as.factor(test.class[,3]))
kbl(linearSVM.conmat$table) %>% kable_styling()
kbl(linearSVM.conmat$overall) %>% kable_styling()
kbl(linearSVM.conmat$byClass) %>% kable_styling()
#          Reference
# Prediction   0   1
#         0  25  16
#         1  35 124

# Accuracy : 0.745
# 95% CI : (0.6787, 0.8039)
# No Information Rate : 0.7
# P-Value [Acc > NIR] : 0.09344

# Kappa : 0.3325
# Mcnemar's Test P-Value : 0.01172

#            Sensitivity : 0.4167
```

```
#            Specificity : 0.8857
#         Pos Pred Value : 0.6098
#         Neg Pred Value : 0.7799
#             Prevalence : 0.3000
#         Detection Rate : 0.1250
#   Detection Prevalence : 0.2050
#      Balanced Accuracy : 0.6512
#       'Positive' Class : 0

## This gives an accuracy of 74.5%. As 51 labels are mispredicted, we can see that the model performs quite well.
## Also, there is a sensitivity of only 41.67% and specificity of ~88.5%.

## Training the model
mdl.linearSVM.class <- train(x = train.class[,1:2], y = as.factor(train.class[,3]),
                             method = "svmLinear", trControl = trainControl("cv", number = 5),
                             tuneGrid = expand.grid(C = seq(0.0, 2, length = 20)))

# Plot model accuracy vs different values of Cost
plot(mdl.linearSVM.class)

# Print the best tuning parameter C that maximises model accuracy
mdl.linearSVM.class$bestTune
#          C
# 15 1.473684



## SVM with radial kernel ##

set.seed(1066)
mdl.radialSVM.class <- train(x = train.class[,1:2], y = as.factor(train.class[,3]), method = "svmRadial")
print(mdl.radialSVM.class)
# 800 samples
# 2 predictors
# 2 classes: '0', '1'

# No pre-processing
# Resampling: Bootstrapped (25 reps)
# Summary of sample sizes: 800, 800, 800, 800, 800, 800, ...
# Resampling results across tuning parameters:

#   C    Accuracy   Kappa
# 0.25  0.8128787  0.5535274
# 0.50  0.8140077  0.5584504
# 1.00  0.8182281  0.5698904

# Tuning parameter 'sigma' was held constant at a value of 1.597849
# Accuracy was used to select the optimal model using the largest value.
# The final values used for the model were sigma = 1.597849 and C = 1.


# Test model on testing data (yTestPred <- mdl %>% predict(xTest))
testPred.radialSVM.class <- predict(mdl.radialSVM.class, newdata = test.class[,1:2])
## predicted vs. true
radialSVM.conmat <- confusionMatrix(as.factor(testPred.radialSVM.class), as.factor(test.class[,3]))
kbl(radialSVM.conmat$table) %>% kable_styling()
kbl(radialSVM.conmat$overall) %>% kable_styling()
```

```
kbl(radialSVM.conmat$byClass) %>% kable_styling()
#          Reference
# Prediction   0   1
#          0  34  11
#          1  26 129

# Accuracy : 0.815
# 95% CI : (0.7541, 0.8663)
# No Information Rate : 0.7
# P-Value [Acc > NIR] : 0.0001479

# Kappa : 0.5256
# Mcnemar's Test P-Value : 0.0213585

#            Sensitivity : 0.5667
#            Specificity : 0.9214
#         Pos Pred Value : 0.7556
#         Neg Pred Value : 0.8323
#             Prevalence : 0.3000
#         Detection Rate : 0.1700
#    Detection Prevalence : 0.2250
#       Balanced Accuracy : 0.7440
#       'Positive' Class : 0

## This gives an accuracy of 81.5%. As 37 labels are mispredicted, we can see that the model performs quite well.
## Also, there is a sensitivity of only 56.67% and specificity of ~92%.

## Training the model
mdl.radialSVM.class <- train(x = train.class[,1:2], y = as.factor(train.class[,3]),
                             method = "svmRadial", trControl = trainControl("cv", number = 5),
                             tuneGrid = expand.grid(C = seq(0.0, 2, length = 20), sigma = 1.597849))

# Plot model accuracy vs different values of Cost
plot(mdl.radialSVM.class)

# Print the best tuning parameter C and sigma that maximises model accuracy
mdl.radialSVM.class$bestTune
#      sigma C
# 3 1.597849 1



## SVM with polynomial kernel ##

set.seed(1066)
mdl.polySVM.class <- train(x = train.class[,1:2], y = as.factor(train.class[,3]), method = "svmPoly")
print(mdl.polySVM.class)
# 800 samples
# 2 predictors
# 2 classes: '0', '1'

# No pre-processing
# Resampling: Bootstrapped (25 reps)
# Summary of sample sizes: 800, 800, 800, 800, 800, 800, ...
# Resampling results across tuning parameters:

# degree  scale  C     Accuracy   Kappa
```

```
# 1      0.001  0.25  0.6729335  0.0000000000
# 1      0.001  0.50  0.6729335  0.0000000000
# 1      0.001  1.00  0.6729335  0.0000000000
# 1      0.010  0.25  0.6729335  0.0000000000
# 1      0.010  0.50  0.6740331  0.0085952848
# 1      0.010  1.00  0.6901900  0.0974061052
# 1      0.100  0.25  0.7233517  0.2629833496
# 1      0.100  0.50  0.7363940  0.3212066422
# 1      0.100  1.00  0.7391340  0.3378417125
# 2      0.001  0.25  0.6729335  0.0000000000
# 2      0.001  0.50  0.6729335  0.0000000000
# 2      0.001  1.00  0.6729335  0.0000000000
# 2      0.010  0.25  0.6744510  0.0098177587
# 2      0.010  0.50  0.6922511  0.1043365054
# 2      0.010  1.00  0.7236850  0.2487301636
# 2      0.100  0.25  0.7803254  0.4417147251
# 2      0.100  0.50  0.7863391  0.4658783475
# 2      0.100  1.00  0.7948156  0.4939772384
# 3      0.001  0.25  0.6729335  0.0000000000
# 3      0.001  0.50  0.6729335  0.0000000000
# 3      0.001  1.00  0.6730705  0.0005488706
# 3      0.010  0.25  0.6836778  0.0568639265
# 3      0.010  0.50  0.7161669  0.2062603369
# 3      0.010  1.00  0.7432817  0.3222420700
# 3      0.100  0.25  0.7905893  0.4772118959
# 3      0.100  0.50  0.8018839  0.5107412544
# 3      0.100  1.00  0.8088213  0.5335867688

# Accuracy was used to select the optimal model using the largest value.
# The final values used for the model were degree = 3, scale = 0.1 and C = 1.

# Test model on testing data (yTestPred <- mdl %>% predict(xTest))
testPred.polySVM.class <- predict(mdl.polySVM.class, newdata = test.class[,1:2])
## predicted vs. true
polySVM.conmat <- confusionMatrix(as.factor(testPred.polySVM.class), as.factor(test.class[,3]))
kbl(polySVM.conmat$table) %>% kable_styling()
kbl(polySVM.conmat$overall) %>% kable_styling()
kbl(polySVM.conmat$byClass) %>% kable_styling()
#          Reference
# Prediction   0   1
#          0  33   9
#          1  27 131

# Accuracy : 0.82
# 95% CI : (0.7596, 0.8706)
# No Information Rate : 0.7
# P-Value [Acc > NIR] : 7.54e-05

# Kappa : 0.5312
# Mcnemar's Test P-Value : 0.004607

#             Sensitivity : 0.5500
#             Specificity : 0.9357
#          Pos Pred Value : 0.7857
#          Neg Pred Value : 0.8291
#              Prevalence : 0.3000
#          Detection Rate : 0.1650
```

```
#    Detection Prevalence : 0.2100
#       Balanced Accuracy : 0.7429
#        'Positive' Class : 0

## This gives an accuracy of 82%. As 36 labels are mispredicted, we can see that the model performs quite well.
## Also, there is a sensitivity of only 55% and specificity of ~93.5%.

## Training the model
mdl.polySVM.class <- train(x = train.class[,1:2], y = as.factor(train.class[,3]),
                           method = "svmPoly", trControl = trainControl("cv", number = 5),
                           tuneGrid = expand.grid(C = seq(0.0, 2, length = 20), scale = 0.1, degree = 3))

# Plot model accuracy vs different values of Cost
plot(mdl.polySVM.class)

# Print the best tuning parameter C, scale and degree that maximises model accuracy
mdl.polySVM.class$bestTune
#    degree scale C
# 27      3   0.1 1



## (e) [8 marks] K-nearest neighbour regression.

## Fit the first KNN model using k = 3 (the chosen k is quite arbitrary here, the sole purpose
## is to show how the fitting procedure works).

## We use the knn function of the class package to fit the model.

## We have to specify the matrix of training predictors (train.class), the matrix of test predictors (test.class),
## the train set labels (cl), and the number of neighbours considered (k).

## The output of the knn function is a vector of predicted labels for the test dataset.

## To check the accuracy of the model we produce the confusion matrix of the model fit using the
## confusionMatrix function of the caret package. This needs the set of predicted labels and the
## set of true test labels as input.

## Note that the confusionMatrix function needs the input to be in factor format. Whenever numbers
## are used as labels, we first have to convert these into factors.

# fit the model (fit = knn(xTrain,xTest,yTrain,k=3))
fit.knn.class = knn(train = train.class[,1:2], test = test.class[,1:2], cl = as.factor(train.class[,3]), k = 3)
fit.knn.class
# produce the confusion matrix
confusionMatrix(as.factor(fit.knn.class), as.factor(test.class[,3]))
# when numbers are used as class labels we might need: confusionMatrix(as.factor(fit),as.factor(yTest))

#          Reference
# Prediction   0   1
#          0  42  18
#          1  18 122

# Accuracy : 0.82
# 95% CI : (0.7596, 0.8706)
# No Information Rate : 0.7
# P-Value [Acc > NIR] : 7.54e-05
```

Kappa : 0.5714
Mcnemar's Test P-Value : 1

Sensitivity : 0.7000
Specificity : 0.8714
Pos Pred Value : 0.7000
Neg Pred Value : 0.8714
Prevalence : 0.3000
Detection Rate : 0.2100
Detection Prevalence : 0.3000
Balanced Accuracy : 0.7857
'Positive' Class : 0

This gives an accuracy of 82%. As 36 labels are mispredicted, we can see that the model performs quite well.
Also, there is a sensitivity of 70% and specificity of ~87%.

The model performance can often be improved by optimising the number of neighbours we use in the fitting procedure.
Here k=3 was arbitrary, but we can use K-fold cross validation to find the optimal number of neighbours.

K-fold cross validation

Machine learning algorithms can deal with non-linearities and complex interactions amongst variables
because the models are flexible enough to fit the data by tuning the model's hyperparameters.

Hyperparameters are parameters that are not directly learnt by the machine learning algorithm,
but affect its structure. For example, consider a simple polynomial regression model:
y = β0 + β1x + β2x2 +···+ βpxp.

The β's are the model parameters that are inferred/learnt from the data. The degree of the polynomial p,
however, is a hyperparameter that dictates the complexity of the model. Hyperparameters are tuned by
cross-validation to strike a balance between underfitting and overfitting, known as the bias-variance trade-off.

In the k-nearest neighbour classification algorithm, k is a hyperparameter.

K-Fold Cross Validation is a technique in which the data set is divided into K-Folds or K-partitions.
The Machine Learning model is trained on K − 1 folds and tested on the Kth fold i.e. we will have K − 1 folds
for training data and 1 for testing the ML model.

The K-Fold Cross Validation estimates the average validation error that we can expect on a new unseen test data.
More precisely, the performance is measured on the part left out in the training process. The average prediction
error is computed from the K runs and the hyperparameters that minimise this error are used to build the final model.

Note that to make cross-validation insensitive to a single random partitioning of the data, repeated cross-validation
is typically performed, where cross-validation is repeated on several random splits of the data.

Repeated cross-validation is what we will use to tune the hyperparameter of the KNN classification.

We build this model on the previously created training dataset. The optimal k for the k-nearest neighbour algorithm
is found using repeated cross validation.

When we want to combine the fitting procedure with the tuning of the hyperparameter, it's not the knn function that we
use anymore. Instead we use the train function of the caret package, which will do the fitting and tuning simultaneously.

The train function sets up a grid of tuning parameters (tuneGrid), these are essentially the k-values we want to try.
It fits the model for each tuning parameter, and calculates a resampling based performance measure. The resampling
method we want to use is given by the trControl option of the train function. Here we will use 10-fold cross validation
(number = 10) repeated 5 times (repeats = 5).

Note that a normal 10-fold cross validation (without repeats) would have trainControl(method = "cv", number = 10)
as trControl parameter.

```
# Set training options
# Repeat 10-fold cross-validation, five times
opts.knn.class <- trainControl(method = "repeatedcv", number = 10, repeats = 5)
# Find optimal k (model)
mdl.knn.class <- train(x = train.class[,1:2],
                       y = as.factor(train.class[,3]),        # training data
                       method = "knn",                    # machine learning model
                       trControl = opts.knn.class,           # training options
                       tuneGrid = data.frame(k = seq(2, 20)))  # range of k's to try

# Plot model accuracy vs different values of Neighbors
plot(mdl.knn.class)

# print the outcome
print(mdl.knn.class)
# 800 samples
# 2 predictors

# No pre-processing
# Resampling: Cross-Validated (10 fold, repeated 5 times)
# Summary of sample sizes: 720, 720, 720, 720, 720, 720, ...
# Resampling results across tuning parameters:

# k   Accuracy   Kappa
# 2   0.7677881  0.4700414
# 3   0.7917854  0.5138814
# 4   0.7907755  0.5122988
# 5   0.8020263  0.5365367
# 6   0.8057951  0.5442054
# 7   0.8037950  0.5387782
# 8   0.8137825  0.5641485
# 9   0.8128046  0.5593758
# 10  0.8142797  0.5621229
# 11  0.8205357  0.5754538
# 12  0.8192890  0.5728601
# 13  0.8225483  0.5789601    # Optimal value
# 14  0.8177891  0.5682497
# 15  0.8155420  0.5638991
# 16  0.8145074  0.5631964
# 17  0.8165172  0.5643226
```

```
# 18  0.8165454  0.5640692
# 19  0.8147607  0.5605534
# 20  0.8132576  0.5574642

# Accuracy was used to select the optimal model using the smallest value.
# The final value used for the model was k = 13.


## This identifies the k that achieves the best accuracy, k=13 in this case.

## As we wanted to see how the cross validation improves the model fit, we only used the previously created
## training data to fit the model with cross validation. Thus we test this model on the test dataset.

## We could now use the knn function to fit a model with k=13 (the optimised parameter), and see its performance
## of test set. An easier approach however is to use the predict function, which takes the previous fitted model
## (mdl.knn.class) as an input an applies that to the dataset specified by the newdata argument. The output is a
## vector of predicted labels for this new dataset.

## Then we use the confusionMatrix function to compare the predicted labels with the true labels, just as before.

# Test model on testing data
test.knn.class.Pred <- predict(mdl.knn.class, newdata = test.class[,1:2])
# predicted vs. true
knn.conmat <- confusionMatrix(as.factor(test.knn.class.Pred), as.factor(test.class[,3]))
kbl(knn.conmat$table) %>% kable_styling()
kbl(knn.conmat$overall) %>% kable_styling()
kbl(knn.conmat$byClass) %>% kable_styling()
#          Reference
# Prediction   0   1
#         0  37  13
#         1  23 127

# Accuracy : 0.82
# 95% CI : (0.7596, 0.8706)
# No Information Rate : 0.7
# P-Value [Acc > NIR] : 7.54e-05

# Kappa : 0.55
# Mcnemar's Test P-Value : 0.1336

#             Sensitivity : 0.6167
#             Specificity : 0.9071
#          Pos Pred Value : 0.7400
#          Neg Pred Value : 0.8467
#              Prevalence : 0.3000
#          Detection Rate : 0.1850
#    Detection Prevalence : 0.2500
#       Balanced Accuracy : 0.7619
#        'Positive' Class : 0

## Now 36 observations of the test dataset are mispredicted (as opposed to 36, when we just picked an arbitrary
## k-value without any tuning), so the accuracy stays the same at 82%. This is of course the test data, which the
model
## was not fitted to, but is used to validate the model instead. This accuracy suggests the model devised from the
## training data was robust.
```

```
## Bonus material ##

## Decision Trees ##

# Fit decision tree
mdl.class <- C5.0(x = train.class[,1:2], y = as.factor(train.class[,3]))

# Plot model decision tree for X1 and X2
plot(mdl.class)

# Predict on test dataset
testPred.class <- predict(mdl.class, newdata = test.class[,1:2])
tree.conmat <- confusionMatrix(as.factor(testPred.class), as.factor(test.class[,3]))
kbl(tree.conmat$table) %>% kable_styling()
kbl(tree.conmat$overall) %>% kable_styling()
kbl(tree.conmat$byClass) %>% kable_styling()
#           Reference
# Prediction   0   1
#         0  26  10
#         1  34 130

# Accuracy : 0.78
# 95% CI : (0.7161, 0.8354)
# No Information Rate : 0.7
# P-Value [Acc > NIR] : 0.0071511

# Kappa : 0.4086
# Mcnemar's Test P-Value : 0.0005256

#            Sensitivity : 0.4333
#            Specificity : 0.9286
#         Pos Pred Value : 0.7222
#         Neg Pred Value : 0.7927
#             Prevalence : 0.3000
#         Detection Rate : 0.1300
#    Detection Prevalence : 0.1800
#       Balanced Accuracy : 0.6810
#       'Positive' Class : 0

## This gives an accuracy of 78%. As 44 labels are mispredicted, we can see that the model performs quite well.
## Also, there is a sensitivity of only 43.33% and specificity of ~93%.


## Random Forest ##

# Fit Random Forest model
# Fix ntree and mtry
set.seed(1066) # for reproducibility
mdl.forest.class <- train(x = train.class[,1:2], y = as.factor(train.class[,3]),
                          method = "rf",
                          ntree = 200,
                          tuneGrid = data.frame(mtry = 100))

# Creating predict function for random forest model and associated ConfusionMatrix
testPred.forest.class <- predict(mdl.forest.class, newdata = test.class[,1:2])
forest.conmat <- confusionMatrix(as.factor(testPred.forest.class), as.factor(test.class[,3]))
kbl(forest.conmat$table) %>% kable_styling()
```

```
kbl(forest.conmat$overall) %>% kable_styling()
kbl(forest.conmat$byClass) %>% kable_styling()
#          Reference
# Prediction   0   1
#          0  40  19
#          1  20 121

# Accuracy : 0.805
# 95% CI : (0.7432, 0.8575)
# No Information Rate : 0.7
# P-Value [Acc > NIR] : 0.0005187

# Kappa : 0.5335
# Mcnemar's Test P-Value : 1.0000000

#             Sensitivity : 0.6667
#             Specificity : 0.8643
#          Pos Pred Value : 0.6780
#          Neg Pred Value : 0.8582
#              Prevalence : 0.3000
#          Detection Rate : 0.2000
#    Detection Prevalence : 0.2950
#       Balanced Accuracy : 0.7655
#       'Positive' Class : 0

## This gives an accuracy of 80.5%. As 39 labels are mispredicted, we can see that the model performs quite well.
## Also, there is a sensitivity of 66.67% and specificity of ~86.5%.

# Variable importance by mean decrease in gini index
varImp(mdl.forest.class$finalModel)
#     Overall
# X1 187.2843
# X2 162.9817



## 4. [3 marks] Compare the results from these five approaches and select what you think is the best method
## for classification in this case, explaining your reasoning.


## LDA

## Print the sensitivity, specificity and cutoff value at the "optimal" value
## sensitivity 0.6571429
## specificity 0.8166667
## cutoff(p)   0.5597111

## The “optimal” point (i.e. point that appears to be closest to the top left-hand corner)
## has a TRP of ~65%, a FPR of ~18% and a cutoff of ~0.56. The AUC is 0.761, which is acceptable.
## This means that 18% of the points that we believe are correct are in the wrong class
## and 65% of points that we believe are correct are actually in the correct class.

## Find the maximum accuracy at the maximum cutoff
## accuracy  cutoff(p)
## 0.7550000 0.5597111

## The accuracy value from the lda accuracy plot at a cutoff (p) of 0.8 is approximately 55%
```

but could be as high as 75.5% at a cutoff(p) of approximately 0.56.

QDA

Print the sensitivity, specificity and cutoff value at the "optimal" value
sensitivity 0.8428571
specificity 0.8000000
cutoff 0.5788973

The “optimal” point (i.e. point that appears to be closest to the top left-hand corner)
has a TRP of ~84%, a FPR of 20% and a cutoff of ~0.58. The AUC is 0.895, which is excellent.
This means that 20% of the points that we believe are correct are in the wrong class
and 84% of points that we believe are correct are actually in the correct class.
The AUC value with qda is great improvement on lda.

Find the maximum accuracy at the maximum cutoff
accuracy cutoff(p)
0.8350000 0.5788973

The accuracy value from the lda accuracy plot at a cutoff (p) of 0.8 is approximately 70%
but could be as high as 83.5% at a cutoff(p) of approximately 0.58.

Logistic regression

From confusionMatrix to compare test set and predictions from the logistic model
(logit.class) to test predicted vs. true
##
Reference
Prediction 0 1
0 36 24
1 31 109

Accuracy : 0.725
Sensitivity : 0.5373
Specificity : 0.8195

From confusionMatrix to compare test data and prediction from
machine-trained model (modelFit.logit.class)
Reference
Prediction 0 1
0 25 35
1 16 124

Accuracy : 0.745
Sensitivity : 0.6098
Specificity : 0.7799

Support vector machines

SVM with linear kernel

Creating ConfusionMatrix for SVM linear kernel (testPred.linearSVM.class)
to test predicted vs. true

```
#          Reference
# Prediction   0   1
#         0  25  16
#         1  35 124

#    Accuracy : 0.745
# Sensitivity : 0.4167
# Specificity : 0.8857
```

This gives an accuracy of 74.5%. As 51 labels are mispredicted, we can see that the model performs quite well.
Also, there is a sensitivity of only 41.67% and specificity of ~88.5%.

SVM with radial kernel

```
# Creating ConfusionMatrix for SVM radial kernel (testPred.radialSVM.class)
# to test predicted vs. true
#          Reference
# Prediction   0   1
#         0  34  11
#         1  26 129

#    Accuracy : 0.815
# Sensitivity : 0.5667
# Specificity : 0.9214
```

This gives an accuracy of 81.5%. As 37 labels are mispredicted, we can see that the model performs quite well.
Also, there is a sensitivity of only 56.67% and specificity of ~92%.

SVM with polynomial kernel

```
# Creating ConfusionMatrix for SVM polynomial kernel (testPred.polySVM.class)
# to test predicted vs. true
#          Reference
# Prediction   0   1
#         0  33   9
#         1  27 131

#    Accuracy : 0.82
# Sensitivity : 0.5500
# Specificity : 0.9357
```

This gives an accuracy of 82%. As 36 labels are mispredicted, we can see that the model performs quite well.
Also, there is a sensitivity of only 55% and specificity of ~93.6%.

K-nearest neighbors

```
# Accuracy was used to select the optimal model using the smallest value.
# The final value used for the model was k = 13.

# Creating ConfusionMatrix for K-nearest neighbours kernel (test.knn.class.Pred)
# to test predicted vs. true
#          Reference
# Prediction   0   1
#         0  37  13
```

1 23 127

Accuracy : 0.82
Sensitivity : 0.6167
Specificity : 0.9071

Now 36 observations of the test dataset are mispredicted (as opposed to 36, when we just picked an arbitrary
k-value without any tuning), so the accuracy stays at 82%. This is of course the test data, which the model
was not fitted to, but is used to validate the model instead. This accuracy suggests the model devised from the
training data was robust.

Bonus material

Decision Tree

From confusionMatrix to compare test data and prediction (testPred.class)
to test predicted vs. true
Reference
Prediction   0   1
0 26 10
1 34 130

Accuracy : 0.78
Sensitivity : 0.4333
Specificity : 0.9286

This gives an accuracy of 78%. As 44 labels are mispredicted, we can see that the model performs quite well.
Also, there is a sensitivity of only 43.33% and specificity of ~93%.

Random Forest

Creating ConfusionMatrix for random forest model (testPred.forest.class)
to test predicted vs. true
Reference
Prediction   0   1
0 40 19
1 20 121

Accuracy : 0.805
Sensitivity : 0.6667
Specificity : 0.8643

This gives an accuracy of 80.5%. As 39 labels are mispredicted, we can see that the model performs quite well.
Also, there is a sensitivity of 66.67% and specificity of ~86.5%.

Comparing results and determining best method

The different methods require different analyses, produce different outputs and use different optimisation methods.
However, the accuracy, sensitivity and specificity results from all the methods can be extracted.

For the two discriminant analyses, QDA produces a ROC curve with a higher AUC value (=0.895) and at the relative
"cutoff" values for each analysis, produces a higher accuracy (=85%) and higher sensitivity (~84%) without losing
too much specificity (=80%).

For the analyses using confusion matrices, the highest accuracy and specificity are from a Support Vector Machine using a
polynomial method (=82% and ~93.6%, respectively), whilst the highest sensitivity is from a Random Forest method (=66.67%).
Overall, the K-nearest neighbours model loses the least with respect to accuracy and specificity (=82% and ~92.1%, respectively)
whilst retaining a high sensitivity of 61.67%.

Therefore, overall, the best method that retains the highest accuracy, sensitivity and specificity is QDA.

5. [4 marks] The file 'ClassificationTrue.csv' contains the true classifications, based on the function of
X1 and X2 without the noise. Evaluate how the 5 different methods from Questions 3 (in each case using the
previously selected optimal value of the parameters) compare to the truth. Does your choice from Question 4
still perform best in this case?

```
# Read in the Classification.csv file and check the header data
true.class <- read.csv("ClassificationTrue.csv")
head(true.class)

# Look at the data
group.true.class <- true.class[,3]
X1.true.class <- true.class[,1]
X2.true.class <- true.class[,2]

# Basic Scatterplot Matrix of X1 and X2
pairs(~ group.true.class + X1.true.class + X2.true.class, data = true.class,
    main = "Simple Scatterplot Matrix of X1 and X2 from True classes")

# Summary statistics of class dataset
summary(true.class)
#       X1                X2              Group
# Min.   :-3.055858   Min.   :-3.41429   Min.   :0.000
# 1st Qu.:-0.679636   1st Qu.:-0.62566   1st Qu.:0.000
# Median :-0.038071   Median : 0.01472   Median :1.000
# Mean   : 0.002033   Mean   : 0.01631   Mean   :0.712
# 3rd Qu.: 0.649037   3rd Qu.: 0.66946   3rd Qu.:1.000
# Max.   : 3.285469   Max.   : 3.94396   Max.   :1.000


# Comparing group.true.class with X1.true.class shows that there is a narrow range between X1.true.class values of -1 to 1
# for group.true.class == 0 and wide range between X1.true.class values of -2 to 2 for group.true.class == 1
# Comparing group.class with X2.true.class shows that there is a wide range between X2.true.class values of -2 to 2
# for group.true.class == 0 and wide range between X2.true.class values of -1 to 3 for group.true.class == 1
# Comparing X1.true.class with X2.true.class shows well scattered data points between -3 and 3 for both parameters
# IQR for both X1.true.class and X2.true.class are approximately between -0.67 and 0.67, with both the mean and median ~ 0.00-0.03.
```

```
# Define function that will split the data into training and test sets
trainTestSplit.true <- function(df.true.class, trainPercent.true, seed3){
  ## Sample size percent
  smp.size.true.class <- floor(trainPercent.true/100 * nrow(df.true.class))
  ## set the seed
  set.seed(seed3)
  train.true <- sample(seq_len(nrow(df.true.class)), size = smp.size.true.class)
  train.true
}

# Split as training and test sets
train.true <- trainTestSplit.true(true.class, trainPercent.true = 80, seed = 321)
train.true.class <- as.data.frame(true.class[train.true, ])
test.true.class <- as.data.frame(true.class[-train.true, ])

count(true.class)
# 1000
length(train.true)
# 800
count(train.true.class)
# 800
count(test.true.class)
# 200


## Linear discriminant analysis.

## Now that our data is ready, we can use the lda() function in R to do our analysis,
## which is functionally identical to the lm() and glm() functions:
f.lda.true.class <- paste(names(train.true.class)[3], "~", paste(names(train.true.class)[-3], collapse = " + "))
true.class.lda <- lda(as.formula(paste(f.lda.true.class)), data = train.true.class)

## Now let's make some predictions on our testing-data:
true.class.lda.predict <- predict(true.class.lda, newdata = test.true.class)

## we can use the lda() function in R to perform our analysis.
lda(as.factor(Group) ~ X1 + X2, data = train.true.class)
##
## Prior probabilities of groups:
##    0    1
## 0.2975 0.7025
##
## Group means:
##        X1         X2
## 0  0.2843937 -0.8468793
## 1 -0.1317206  0.3901919
##
## Coefficients of linear discriminants:
##        LD1
## X1 -0.3545344
## X2  1.1331036

## Finally we can look at the predictions and compare them to the test/evaluation dataset just
## as we did for other methods.

## Evaluation and construction of ROC AUC plot:
```

```
## Get the posteriors as a dataframe.
true.class.lda.predict.posteriors <- as.data.frame(true.class.lda.predict$posterior)
true.class.lda.predict.posteriors
## Evaluate the model
pred.lda.true.class <- prediction(true.class.lda.predict.posteriors[,2], as.factor(test.true.class[,3]))
roc.perf.lda.true.class <- performance(pred.lda.true.class, measure = "tpr", x.measure = "fpr")
auc.train.lda.true.class <- performance(pred.lda.true.class, measure = "auc")
auc.train.lda.true.class <- auc.train.lda.true.class@y.values
## Plot
plot(roc.perf.lda.true.class)
abline(a = 0, b = 1)
text(x = 0.25, y = 0.65, paste("AUC = ", round(auc.train.lda.true.class[[1]], 3), sep = ""))

## To find the optimal cutoff point
opt.cut.true.lda <- function(roc.perf.lda.true.class, pred.lda.true.class){
  cut.ind.true.lda <- mapply(FUN = function(x, y, p){
    d = (x - 0)^2 + (y-1)^2
    true.ind = which(d == min(d))
    c(sensitivity = y[[true.ind]], specificity = 1-x[[true.ind]],
      cutoff = p[[true.ind]])
  }, roc.perf.lda.true.class@x.values, roc.perf.lda.true.class@y.values, pred.lda.true.class@cutoffs)
}

## Print the sensitivity, specificity and cutoff value at the "optimal" value
print(opt.cut.true.lda(roc.perf.lda.true.class, pred.lda.true.class))
## sensitivity 0.7400000
## specificity 0.7800000
## cutoff(p)   0.4019321

## The “optimal” point (i.e. point that appears to be closest to the top left-hand corner)
## has a TRP of 74%, a FPR of 22% and a cutoff of ~0.70. The AUC is 0.816, which is good.
## This means that 22% of the points that we believe are correct are in the wrong class
## and 74% of points that we believe are correct are actually in the correct class.

## Also, let's get the overall accuracy for the lda predictions and plot it:
acc.lda.true.perf <- performance(pred.lda.true.class, measure = "acc")
plot(acc.lda.true.perf)

## And find the maximum accuracy at the maximum cutoff
ind.true.lda <- which.max(slot(acc.lda.true.perf, "y.values")[[1]])
lda.true.acc <- slot(acc.lda.true.perf, "y.values")[[1]][ind.true.lda]
cutoff.true.lda <- slot(acc.lda.true.perf, "x.values")[[1]][ind.true.lda]
print(c(accuracy = lda.true.acc, cutoff = cutoff.true.lda))
## accuracy  cutoff(p)
## 0.7850000 0.4019321

## The accuracy value from the lda accuracy plot at a cutoff (p) of 0.8 is approximately 66%
## but could be as high as 78.5% at a cutoff(p) of approximately 0.4.



## (b) Quadratic discriminant analysis.

## Now that our data is ready, we can use the qda() function in R to do our analysis:
f.qda.true.class <- paste(names(train.true.class)[3], "~", paste(names(train.true.class)[-3], collapse = " + "))
true.class.qda <- qda(as.formula(paste(f.qda.true.class)), data = train.true.class)
```

```
## Now let's make some predictions on our testing-data:
true.class.qda.predict <- predict(true.class.qda, newdata = test.true.class)

## we can use the qda() function in R to perform our analysis.
qda(as.factor(Group) ~ X1 + X2, data = train.true.class)
##
## Prior probabilities of groups:
##    0    1
## 0.2975 0.7025
##
## Group means:
##        X1         X2
## 0  0.2843937 -0.8468793
## 1 -0.1317206  0.3901919
##

## Finally we can look at the predictions and compare them to the test/evaluation dataset just
## as we did for other methods.

## Evaluation and construction of ROC AUC plot:
## Get the posteriors as a dataframe.
true.class.qda.predict.posteriors <- as.data.frame(true.class.qda.predict$posterior)
true.class.qda.predict.posteriors
## Evaluate the model
pred.qda.true.class <- prediction(true.class.qda.predict.posteriors[,2], as.factor(test.true.class[,3]))
roc.perf.qda.true.class <- performance(pred.qda.true.class, measure = "tpr", x.measure = "fpr")
auc.train.qda.true.class <- performance(pred.qda.true.class, measure = "auc")
auc.train.qda.true.class <- auc.train.qda.true.class@y.values
## Plot
plot(roc.perf.qda.true.class)
abline(a = 0, b = 1)
text(x = 0.25, y = 0.65, paste("AUC = ", round(auc.train.qda.true.class[[1]], 3), sep = ""))

## To find the optimal cutoff point
opt.cut.true.qda <- function(roc.perf.qda.true.class, pred.qda.true.class){
  cut.ind.true.qda <- mapply(FUN = function(x, y, p){
    d = (x - 0)^2 + (y-1)^2
    true.ind = which(d == min(d))
    c(sensitivity = y[[true.ind]], specificity = 1-x[[true.ind]],
      cutoff = p[[true.ind]])
  }, roc.perf.qda.true.class@x.values, roc.perf.qda.true.class@y.values, pred.qda.true.class@cutoffs)
}

## Print the sensitivity, specificity and cutoff value at the "optimal" value
print(opt.cut.true.qda(roc.perf.qda.true.class, pred.qda.true.class))
## sensitivity 0.8400000
## specificity 0.9000000
## cutoff      0.4947414

## The "optimal" point (i.e. point that appears to be closest to the top left-hand corner)
## has a TRP of 84%, a FPR of 10% and a cutoff of ~0.495. The AUC is 0.945, which is excellent.
## This means that 10% of the points that we believe are correct are in the wrong class
## and 84% of points that we believe are correct are actually in the correct class.
## The AUC value with qda is great improvement on lda.

## Finally, let's get the overall accuracy for the qda predictions and plot it:
acc.qda.true.perf = performance(pred.qda.true.class, measure = "acc")
```

```
plot(acc.qda.true.perf)

## And find the maximum accuracy at the maximum cutoff
ind.true.qda <- which.max(slot(acc.qda.true.perf, "y.values")[[1]])
qda.true.acc <- slot(acc.qda.true.perf, "y.values")[[1]][ind.true.qda]
cutoff.true.qda <- slot(acc.qda.perf, "x.values")[[1]][ind.true.qda]
print(c(accuracy = qda.true.acc, cutoff = cutoff.true.qda))
## accuracy  cutoff(p)
## 0.8950000 0.4947414

## The accuracy value from the lda accuracy plot at a cutoff (p) of 0.8 is approximately 82%
## but could be as high as 89.5% at a cutoff(p) of approximately 0.495.



## (c) Logistic regression.

## Next let's start by fitting a logistic regression will all the variables using the
## glm() function, with family = 'binomial'.

## Here we fit a logistic model to the data
logit.true.class <- glm(as.factor(Group) ~ X1 + X2, family = "binomial", data = train.true.class)

# Plot logistic model residuals vs. fitted, normal qq, scale-location and residual vs. leverage plots
par(mar = c(2,4,4,2), cex = 1.0)
layout(matrix(c(1,1,2,2),2,2,byrow=TRUE))
plot(logit.true.class)

# Look at the summary of the logistic model
summary(logit.true.class)
##
## Deviance Residuals:
##    Min      1Q     Median      3Q     Max
##  -2.2723  -0.6043   0.2918   0.6047   2.9036
##
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept)  1.3758     0.1160  11.863   < 2e-16 ***
## X1          -0.5805     0.1072  -5.414  6.15e-08 ***
## X2           1.9095     0.1477  12.932   < 2e-16 ***
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 973.97 on 799  degrees of freedom
## Residual deviance: 639.22 on 797  degrees of freedom
## AIC: 645.22
##
## Number of Fisher Scoring iterations: 5

# Run the model on the test set
test.true.class.probs <- predict(logit.true.class, newdata = test.true.class[,1:2])
pred.logit.true.class <- rep(0, length(test.true.class.probs))
pred.logit.true.class[test.true.class.probs >= 0.5] <- 1

# Create table to compare test set and predictions
table(pred.logit.true.class, test.true.class$Group)
##
```

```
## pred.logit.true.class   0   1
##                   0  29  29
##                   1  21 121

# Set up ConfusionMatrix to compare test set and predictions from the logistic model
confusionMatrix(as.factor(test.true.class[,3]), as.factor(pred.logit.true.class))
## Confusion Matrix and Statistics
##
##          Reference
## Prediction   0   1
##         0  29  21
##         1  29 121

## Accuracy : 0.75
## 95% CI : (0.684, 0.8084)
## No Information Rate : 0.71
## P-Value [Acc > NIR] : 0.1203

## Kappa : 0.3671
## Mcnemar's Test P-Value : 0.3222

##            Sensitivity : 0.5000
##            Specificity : 0.8521
##         Pos Pred Value : 0.5800
##         Neg Pred Value : 0.8067
##             Prevalence : 0.2900
##         Detection Rate : 0.1450
##    Detection Prevalence : 0.2500
##       Balanced Accuracy : 0.6761
##       'Positive' Class : 0
##

## This gets 50 points incorrect (out of 200), so has an accuracy of 75%, with a sensitivity of only 50%
## and specificity of ~85%. This is good but the caret package will be used create a machine-learned logistic
## model to attempt to improve on this accuracy.


# Use the fitted model to get the probability that a given obs belongs to class 1 using the predict function.
true.class.logit.predict <- predict(logit.true.class, type = "response")

# In order to use the geom_roc function we have to create a dataframe that contains the estimated
# probabilities and the actual class labels.

# Create a dataframe that has the true labels, and the probabilities that the glm fit gives for
# belonging to class 1 by simulating some data
df.logit.true.class <- data.frame(a = test.true.class[,3], b = predict(logit.true.class, type = "response"))

# Use ggplot with the geom_roc option to get the ROC curve. The asthetics of geom_roc are
# the true class labels, d, and the associated probabilities of belonging to class 1, m.
ROC.logit.true.class <- ggplot(df.logit.true.class, aes(d = a, m = b)) +
  geom_roc() + ylab("Sensitivity") + xlab("1-Specificity")
ROC.logit.true.class

# Finally, we can get the area under the receiver operating curve by using the calc_auc function
# on the previous ggplot object - It's actually worse than random chance (AUC < 0.5).
calc_auc(ROC.logit.true.class)$AUC
# [1] 0.49205
```

Comparing the two optimisation methods, there appears to be a contrast in results based on using a Confusion Matrix,
which gives an accuracy of 75%, a sensitivity of 50% and a specificity of ~85%, and using a ROC curve,
with an AUC value of 0.49 (suggesting a worse result than random chance). Consequently, the result from the
Confusion Matrix is the more reliable (i.e. "less confused") method and is preferred here.

Now use the caret package to train the machine to fit a logistic model
```
modelFit.logit.true.class <- train(as.factor(Group) ~ X1 + X2, data = train.true.class,
                        method = "glm", family = binomial(link = "logit"),
                        preProcess = c('scale', 'center'))

# Look at the summary of the machine-trained model
summary(modelFit.logit.true.class)
```
##
Deviance Residuals:
Min   1Q   Median   3Q   Max
-2.2723 -0.6043   0.2918   0.6047   2.9036
##
Coefficients:
Estimate Std. Error z value Pr(>|z|)
(Intercept) 1.4228   0.1179 12.071   < 2e-16 ***
X1   -0.5620   0.1038 -5.414 6.15e-08 ***
X2   1.9456   0.1504 12.932   < 2e-16 ***
##
(Dispersion parameter for binomial family taken to be 1)
##
Null deviance: 973.97 on 799 degrees of freedom
Residual deviance: 639.22 on 797 degrees of freedom
AIC: 645.22
##
Number of Fisher Scoring iterations: 5

Create confusionMatrix to compare test data and prediction from the machine-trained model
```
confusionMatrix(as.factor(test.true.class[,3]), as.factor(predict(modelFit.logit.true.class, test.true.class)))
```
##
Reference
Prediction   0   1
0 25 25
1 22 128

Accuracy : 0.765
95% CI : (0.7, 0.8219)
No Information Rate : 0.765
P-Value [Acc > NIR] : 0.5390

Kappa : 0.3605
Mcnemar's Test P-Value : 0.7705

Sensitivity : 0.5319
Specificity : 0.8366
Pos Pred Value : 0.5000
Neg Pred Value : 0.8533
Prevalence : 0.2350

```
##         Detection Rate : 0.1250
##   Detection Prevalence : 0.2500
##      Balanced Accuracy : 0.6843
##       'Positive' Class : 0

## The machine-trained model obtains about 76.5% accuracy overall (47 of 200 points were incorrect),
## with a sensitivity of only 53.2% and specificity of ~83.7%. This is still good and is also an
## improvement on the 75% prediction accuracy from the initial logistic model. This suggests that
## a machine can be trained to create a model with better accuracy.


# Use the fitted model to get the probability that a given obs belongs to class 1 using the predict function.
predict(modelFit.logit.true.class, type = "raw")

# In order to use the geom_roc function we have to create a dataframe that contains the estimated
# probabilities and the actual class labels.

# Create a dataframe that has the true labels, and the probabilities that the glm fit gives for
# belonging to class 1 by simulating some data
df.modelFit.logit.true.class <- data.frame(a = test.true.class[,3], b = predict(modelFit.logit.true.class, type = "raw"))

# Use ggplot with the geom_roc option to get the ROC curve. The asthetics of geom_roc are
# the true class labels, d, and the associated probabilities of belonging to class 1, m.
ROC.modelFit.logit.true.class <- ggplot(df.modelFit.logit.true.class, aes(d = a, m = b)) +
  geom_roc() + ylab("Sensitivity") + xlab("1-Specificity")
ROC.modelFit.logit.true.class

# Finally, we can get the area under the receiver operating curve by using the calc_auc function
# on the previous ggplot object - It's actually worse than random chance (AUC < 0.5).
calc_auc(ROC.modelFit.logit.true.class)$AUC
# [1] 0.5



## (d) Support vector machines.


## SVM with linear kernel ##

mdl.linearSVM.true.class <- train(x = train.true.class[,1:2], y = as.factor(train.true.class[,3]), method = "svmLinear")
print(mdl.linearSVM.true.class)
# 800 samples
# 2 predictors
# 2 classes: '0', '1'

# No pre-processing
# Resampling: Bootstrapped (25 reps)
# Summary of sample sizes: 800, 800, 800, 800, 800, 800, ...
# Resampling results:

#  Accuracy   Kappa
# 0.812843  0.5347685

# Tuning parameter 'C' was held constant at a value of 1


# Test model on testing data (yTestPred <- mdl %>% predict(xTest))
```

```
testPred.linearSVM.true.class <- predict(mdl.linearSVM.true.class, newdata = test.true.class[,1:2])
# predicted vs. true
confusionMatrix(as.factor(testPred.linearSVM.true.class), as.factor(test.true.class[,3]))
#          Reference
# Prediction   0   1
#         0  25  22
#         1  25 128

# Accuracy : 0.765
# 95% CI : (0.7, 0.8219)
# No Information Rate : 0.75
# P-Value [Acc > NIR] : 0.3458

# Kappa : 0.3605
# Mcnemar's Test P-Value : 0.7705

#            Sensitivity : 0.5000
#            Specificity : 0.8533
#         Pos Pred Value : 0.5319
#         Neg Pred Value : 0.8366
#             Prevalence : 0.2500
#         Detection Rate : 0.1250
#    Detection Prevalence : 0.2350
#       Balanced Accuracy : 0.6767
#       'Positive' Class : 0

## This gives an accuracy of 76.5%. As 47 labels are mispredicted, we can see that the model performs quite well.
## Also, there is a sensitivity of only 50% and specificity of ~85.33%.

## Training the model
mdl.linearSVM.true.class <- train(x = train.true.class[,1:2], y = as.factor(train.true.class[,3]),
                        method = "svmLinear", trControl = trainControl("cv", number = 5),
                        tuneGrid = expand.grid(C = seq(0.0, 2, length = 20)))

# Plot model accuracy vs different values of Cost
plot(mdl.linearSVM.true.class)

# Print the best tuning parameter C that maximises model accuracy
mdl.linearSVM.true.class$bestTune
#          C
# 2 0.1052632


## SVM with radial kernel ##

mdl.radialSVM.true.class <- train(x = train.true.class[,1:2], y = as.factor(train.true.class[,3]), method = "svmRadial")
print(mdl.radialSVM.true.class)
# 800 samples
# 2 predictors
# 2 classes: '0', '1'

# No pre-processing
# Resampling: Bootstrapped (25 reps)
# Summary of sample sizes: 800, 800, 800, 800, 800, 800, ...
# Resampling results across tuning parameters:

#   C    Accuracy   Kappa
```

```
# 0.25  0.9022654  0.7575127
# 0.50  0.9033900  0.7616415
# 1.00  0.9052671  0.7663391

# Tuning parameter 'sigma' was held constant at a value of 1.018684
# Accuracy was used to select the optimal model using the largest value.
# The final values used for the model were sigma = 1.018684 and C = 1.


# Test model on testing data (yTestPred <- mdl %>% predict(xTest))
testPred.radialSVM.true.class <- predict(mdl.radialSVM.true.class, newdata = test.true.class[,1:2])
## predicted vs. true
confusionMatrix(as.factor(testPred.radialSVM.true.class), as.factor(test.true.class[,3]))
#          Reference
# Prediction   0   1
#          0  36  11
#          1  14 139

# Accuracy : 0.875
# 95% CI : (0.821, 0.9174)
# No Information Rate : 0.75
# P-Value [Acc > NIR] : 9.355e-06

# Kappa : 0.6599
# Mcnemar's Test P-Value : 0.6892

#             Sensitivity : 0.7200
#             Specificity : 0.9267
#          Pos Pred Value : 0.7660
#          Neg Pred Value : 0.9085
#              Prevalence : 0.2500
#          Detection Rate : 0.1800
#    Detection Prevalence : 0.2350
#       Balanced Accuracy : 0.8233
#       'Positive' Class : 0

## This gives an accuracy of 87.5%. As 25 labels are mispredicted, we can see that the model performs quite well.
## Also, there is a sensitivity of only 72% and specificity of ~92.67%.

## Training the model
mdl.radialSVM.true.class <- train(x = train.true.class[,1:2], y = as.factor(train.true.class[,3]),
                                  method = "svmRadial", trControl = trainControl("cv", number = 5),
                                  tuneGrid = expand.grid(C = seq(0.0, 2, length = 20), sigma = 1.018684))

# Plot model accuracy vs different values of Cost
plot(mdl.radialSVM.true.class)

# Print the best tuning parameter C and sigma that maximises model accuracy
mdl.radialSVM.true.class$bestTune
#    sigma       C
# 18 1.018684 1.789474



## SVM with polynomial kernel ##

mdl.polySVM.true.class <- train(x = train.true.class[,1:2], y = as.factor(train.true.class[,3]), method = "svmPoly")
```

```
print(mdl.polySVM.true.class)
# 800 samples
# 2 predictors
# 2 classes: '0', '1'

# No pre-processing
# Resampling: Bootstrapped (25 reps)
# Summary of sample sizes: 800, 800, 800, 800, 800, 800, ...
# Resampling results across tuning parameters:

# degree  scale  C     Accuracy   Kappa
# 1       0.001  0.25  0.7042142  0.0000000000
# 1       0.001  0.50  0.7042142  0.0000000000
# 1       0.001  1.00  0.7042142  0.0000000000
# 1       0.010  0.25  0.7053032  0.0072708738
# 1       0.010  0.50  0.7361624  0.1919127561
# 1       0.010  1.00  0.7958366  0.4464958559
# 1       0.100  0.25  0.8111396  0.5198456840
# 1       0.100  0.50  0.8126897  0.5286771352
# 1       0.100  1.00  0.8133872  0.5320629962
# 2       0.001  0.25  0.7042142  0.0000000000
# 2       0.001  0.50  0.7042142  0.0000000000
# 2       0.001  1.00  0.7043416  0.0007285234
# 2       0.010  0.25  0.7380745  0.1973163860
# 2       0.010  0.50  0.7978341  0.4513477517
# 2       0.010  1.00  0.8154698  0.5266326549
# 2       0.100  0.25  0.8651504  0.6559430780
# 2       0.100  0.50  0.8747111  0.6858798704
# 2       0.100  1.00  0.8851324  0.7178102706
# 3       0.001  0.25  0.7042142  0.0000000000
# 3       0.001  0.50  0.7042142  0.0000000000
# 3       0.001  1.00  0.7096882  0.0318847574
# 3       0.010  0.25  0.7810312  0.3825094796
# 3       0.010  0.50  0.8139539  0.5151101535
# 3       0.010  1.00  0.8246180  0.5532573444
# 3       0.100  0.25  0.8824855  0.7070639148
# 3       0.100  0.50  0.8921829  0.7359931369
# 3       0.100  1.00  0.8995425  0.7571663227

# Accuracy was used to select the optimal model using the largest value.
# The final values used for the model were degree = 3, scale = 0.1 and C = 1.

# Test model on testing data (yTestPred <- mdl %>% predict(xTest))
testPred.polySVM.true.class <- predict(mdl.polySVM.true.class, newdata = test.true.class[,1:2])
## predicted vs. true
confusionMatrix(as.factor(testPred.polySVM.true.class), as.factor(test.true.class[,3]))
#          Reference
# Prediction   0   1
#          0  34  11
#          1  16 139

# Accuracy : 0.865
# 95% CI : (0.8097, 0.9091)
# No Information Rate : 0.75
# P-Value [Acc > NIR] : 4.813e-05

# Kappa : 0.6276
```

```
# Mcnemar's Test P-Value : 0.4414

#            Sensitivity : 0.6800
#            Specificity : 0.9267
#         Pos Pred Value : 0.7556
#         Neg Pred Value : 0.8968
#             Prevalence : 0.2500
#         Detection Rate : 0.1700
#   Detection Prevalence : 0.2250
#      Balanced Accuracy : 0.8033
#       'Positive' Class : 0

## This gives an accuracy of 86.5%. As 27 labels are mispredicted, we can see that the model performs quite well.
## Also, there is a sensitivity of 68% and specificity of 92.67%.

## Training the model
mdl.polySVM.true.class <- train(x = train.true.class[,1:2], y = as.factor(train.true.class[,3]),
                    method = "svmPoly", trControl = trainControl("cv", number = 5),
                    tuneGrid = expand.grid(C = seq(0.0, 2, length = 20), scale = 0.1, degree = 3))

# Plot model accuracy vs different values of Cost
plot(mdl.polySVM.true.class)

# Print the best tuning parameter C, scale and degree that maximises model accuracy
mdl.polySVM.true.class$bestTune
#    degree scale     C
# 18      3   0.1 1.789474



## (e) K-nearest neighbour regression.

## Fit the first KNN model using k = 3 (the chosen k is quite arbitrary here, the sole purpose
## is to show how the fitting procedure works).

## We use the knn function of the class package to fit the model.

## We have to specify the matrix of training predictors (train.class), the matrix of test predictors (test.class),
## the train set labels (cl), and the number of neighbours considered (k).

## The output of the knn function is a vector of predicted labels for the test dataset.

## To check the accuracy of the model we produce the confusion matrix of the model fit using the
## confusionMatrix function of the caret package. This needs the set of predicted labels and the
## set of true test labels as input.

## Note that the confusionMatrix function needs the input to be in factor format. Whenever numbers
## are used as labels, we first have to convert these into factors.

# fit the model (fit = knn(xTrain,xTest,yTrain,k=3))
fit.knn.true.class = knn(train = train.true.class[,1:2], test = test.true.class[,1:2], cl = as.factor(train.true.class[,3]), k =
3)
fit.knn.true.class
# produce the confusion matrix
confusionMatrix(as.factor(fit.knn.true.class), as.factor(test.true.class[,3]))
# when numbers are used as class labels we might need: confusionMatrix(as.factor(fit),as.factor(yTest))
#          Reference
```

```
# Prediction   0   1
#          0  36  11
#          1  14 139

# Accuracy : 0.875
# 95% CI : (0.821, 0.9174)
# No Information Rate : 0.75
# P-Value [Acc > NIR] : 9.355e-06

# Kappa : 0.6599
# Mcnemar's Test P-Value : 0.6892

#             Sensitivity : 0.7200
#             Specificity : 0.9267
#          Pos Pred Value : 0.7660
#          Neg Pred Value : 0.9085
#              Prevalence : 0.2500
#          Detection Rate : 0.1800
#    Detection Prevalence : 0.2350
#       Balanced Accuracy : 0.8233
#       'Positive' Class : 0
```

This gives an accuracy of 87.5%. As 25 labels are mispredicted, we can see that the model performs quite well.
Also, there is a sensitivity of 72% and specificity of ~92.67%.

The model performance can often be improved by optimising the number of neighbours we use in the fitting procedure.
Here k=3 was arbitrary, but we can use K-fold cross validation to find the optimal number of neighbours.

K-fold cross validation

Machine learning algorithms can deal with non-linearities and complex interactions amongst variables
because the models are flexible enough to fit the data by tuning the model's hyperparameters.

Hyperparameters are parameters that are not directly learnt by the machine learning algorithm,
but affect its structure. For example, consider a simple polynomial regression model:
y = β0 + β1x + β2x2 +···+ βpxp.

The β's are the model parameters that are inferred/learnt from the data. The degree of the polynomial p,
however, is a hyperparameter that dictates the complexity of the model. Hyperparameters are tuned by
cross-validation to strike a balance between underfitting and overfitting, known as the bias-variance trade-off.

In the k-nearest neighbour classification algorithm, k is a hyperparameter.

K-Fold Cross Validation is a technique in which the data set is divided into K-Folds or K-partitions.
The Machine Learning model is trained on K − 1 folds and tested on the Kth fold i.e. we will have K − 1 folds
for training data and 1 for testing the ML model.

The K-Fold Cross Validation estimates the average validation error that we can expect on a new unseen test data.
More precisely, the performance is measured on the part left out in the training process. The average prediction
error is computed from the K runs and the hyperparameters that minimise this error are used to build the final model.

Note that to make cross-validation insensitive to a single random partitioning of the data, repeated cross-validation
is typically performed, where cross-validation is repeated on several random splits of the data.

Repeated cross-validation is what we will use to tune the hyperparameter of the KNN classification.

We build this model on the previously created training dataset. The optimal k for the k-nearest neighbour algorithm
is found using repeated cross validation.

When we want to combine the fitting procedure with the tuning of the hyperparameter, it's not the knn function that we
use anymore. Instead we use the train function of the caret package, which will do the fitting and tuning simultaneously.

The train function sets up a grid of tuning parameters (tuneGrid), these are essentially the k-values we want to try.
It fits the model for each tuning parameter, and calculates a resampling based performance measure. The resampling
method we want to use is given by the trControl option of the train function. Here we will use 10-fold cross validation
(number = 10) repeated 5 times (repeats = 5).

Note that a normal 10-fold cross validation (without repeats) would have trainControl(method = "cv", number = 10)
as trControl parameter.

```
# Set training options
# Repeat 10-fold cross-validation, five times
opts.knn.true.class <- trainControl(method = "repeatedcv", number = 10, repeats = 5)
# Find optimal k (model)
mdl.knn.true.class <- train(x = train.true.class[,1:2],
                            y = as.factor(train.true.class[,3]),    # training data
                            method = "knn",                         # machine learning model
                            trControl = opts.knn.true.class,        # training options
                            tuneGrid = data.frame(k = seq(2, 20)))  # range of k's to try

# Plot model accuracy vs different values of Neighbors
plot(mdl.knn.true.class)

# print the outcome
print(mdl.knn.true.class)
# 800 samples
# 2 predictors

# No pre-processing
# Resampling: Cross-Validated (10 fold, repeated 5 times)
# Summary of sample sizes: 720, 720, 720, 720, 720, 720, ...
# Resampling results across tuning parameters:

# k   Accuracy   Kappa
# 2  0.8752990  0.7003287
# 3  0.8908282  0.7359833
# 4  0.8850495  0.7241308
# 5  0.8905719  0.7367549
# 6  0.8858276  0.7264956
# 7  0.8998005  0.7571066
# 8  0.8985690  0.7548623
# 9  0.9020817  0.7632578     # Optimal value
# 10  0.8998440  0.7591928
```

```
# 11  0.8993438  0.7575491
# 12  0.8998377  0.7593179
# 13  0.8990907  0.7583238
# 14  0.8983408  0.7563839
# 15  0.8955969  0.7494157
# 16  0.8948470  0.7475989
# 17  0.9010724  0.7622317
# 18  0.8968251  0.7521117
# 19  0.8980784  0.7558892
# 20  0.8980877  0.7564715

# Accuracy was used to select the optimal model using the smallest value.
# The final value used for the model was k = 9.


## This identifies the k that achieves the best accuracy, k=9 in this case.

## As we wanted to see how the cross validation improves the model fit, we only used the previously created
## training data to fit the model with cross validation. Thus we test this model on the test dataset.

## We could now use the knn function to fit a model with k=13 (the optimised parameter), and see its performance
## of test set. An easier approach however is to use the predict function, which takes the previous fitted model
## (mdl.knn.class) as an input an applies that to the dataset specified by the newdata argument. The output is a
## vector of predicted labels for this new dataset.

## Then we use the confusionMatrix function to compare the predicted labels with the true labels, just as before.

# Test model on testing data
test.knn.true.class.Pred <- predict(mdl.knn.true.class, newdata = test.true.class[,1:2])
# predicted vs. true
confusionMatrix(as.factor(test.knn.true.class.Pred), as.factor(test.true.class[,3]))
#          Reference
# Prediction   0   1
#          0  33  11
#          1  17 139

# Accuracy : 0.86
# 95% CI : (0.8041, 0.9049)
# No Information Rate : 0.75
# P-Value [Acc > NIR] : 0.0001026

# Kappa : 0.6111
# Mcnemar's Test P-Value : 0.3447042

#             Sensitivity : 0.6600
#             Specificity : 0.9267
#          Pos Pred Value : 0.7500
#          Neg Pred Value : 0.8910
#              Prevalence : 0.2500
#          Detection Rate : 0.1650
#    Detection Prevalence : 0.2200
#       Balanced Accuracy : 0.7933
#       'Positive' Class : 0

## Now 28 observations of the test dataset are mispredicted (as opposed to 25, when we just picked an arbitrary
## k-value without any tuning) and the accuracy has dropped to 86%. This is of course the test data, which the model
## was not fitted to, but is used to validate the model instead. This accuracy suggests the model devised from the
```

```
## training data was robust.



## Bonus material ##

## Decision Trees ##

# Fit decision tree
mdl.true.class <- C5.0(x = train.true.class[,1:2], y = as.factor(train.true.class[,3]))

# Plot model decision tree for X1 and X2
plot(mdl.true.class)

# Predict on test dataset
testPred.true.class <- predict(mdl.true.class, newdata = test.true.class[,1:2])
confusionMatrix(as.factor(testPred.true.class), as.factor(test.true.class[,3]))
#          Reference
# Prediction   0   1
#         0  34   7
#         1  16 143

# Accuracy : 0.885
# 95% CI : (0.8325, 0.9257)
# No Information Rate : 0.75
# P-Value [Acc > NIR] : 1.527e-06

# Kappa : 0.6738
# Mcnemar's Test P-Value : 0.09529

#            Sensitivity : 0.6800
#            Specificity : 0.9533
#         Pos Pred Value : 0.8293
#         Neg Pred Value : 0.8994
#             Prevalence : 0.2500
#         Detection Rate : 0.1700
#    Detection Prevalence : 0.2050
#       Balanced Accuracy : 0.8167
#       'Positive' Class : 0

## This gives an accuracy of 88.5%. As only 23 labels are mispredicted, we can see that the model performs well.
## Also, there is a sensitivity of 68% and specificity of ~95.3%.



## Random Forest ##

# Fit Random Forest model
# Fix ntree and mtry
set.seed(1066) # for reproducibility
mdl.forest.true.class <- train(x = train.true.class[,1:2],
                               y = as.factor(train.true.class[,3]),
                               method = "rf",
                               ntree = 200,
                               tuneGrid = data.frame(mtry = 100))

# Creating predict function for random forest model and associated ConfusionMatrix
testPred.forest.true.class <- predict(mdl.forest.true.class, newdata = test.true.class[,1:2])
```

```
confusionMatrix(as.factor(testPred.forest.true.class), as.factor(test.true.class[,3]))
#          Reference
# Prediction   0   1
#          0  38  10
#          1  12 140

# Accuracy : 0.89
# 95% CI : (0.8382, 0.9298)
# No Information Rate : 0.75
# P-Value [Acc > NIR] : 5.752e-07

# Kappa : 0.7027
# Mcnemar's Test P-Value : 0.8312

#            Sensitivity : 0.7600
#            Specificity : 0.9333
#         Pos Pred Value : 0.7917
#         Neg Pred Value : 0.9211
#             Prevalence : 0.2500
#         Detection Rate : 0.1900
#    Detection Prevalence : 0.2400
#       Balanced Accuracy : 0.8467
#       'Positive' Class : 0

## This gives an accuracy of 89%. As 22 labels are mispredicted, we can see that the model performs quite well.
## Also, there is a sensitivity of 76% and specificity of ~93.3%.

# Variable importance by mean decrease in gini index
varImp(mdl.forest.true.class$finalModel)
#     Overall
# X1 166.0949
# X2 168.8317
```

Comparison of results from data in Classification.csv and ClassificationTrue.csv

LDA - The "True" dataset displays improved accuracy (=78%) and sensitivity (=74%),
with a minor reduction in specificity (=78%) compared with the initial dataset.
There is also an improvement in AUC in the ROC curve to 0.816.

QDA - The "True" dataset displays improved accuracy (=89.5%) and specificity (=90%),
with the same sensitivity (=84%) compared with the initial dataset.
There is also an improvement in AUC in the ROC curve to 0.945.

Logistic regression (glm) - The "True" dataset displays improved accuracy (=75%) and specificity (=53%),
with a slight reduction in sensitivity (=50%) compared with the initial dataset.

Logistic regression (ml) - The "True" dataset displays improved accuracy (=76.5%), specificity (~83.7%),
with a slight reduction in sensitivity (~53.2%) compared with the initial dataset.

Support vector machines

Decision Tree - The "True" dataset displays improved accuracy (=88.5%), specificity (~95.3%),
and also in sensitivity (68%) compared with the initial dataset.

Random Forest - The "True" dataset displays improved accuracy (=89%), specificity (~93.3%),
and also in sensitivity (76%) compared with the initial dataset.

SVM with linear kernel - The "True" dataset displays improved accuracy (=76.5%), specificity (~85.3%),
and also in sensitivity (50%) compared with the initial dataset.## Also, there is a sensitivity of only 50% and specificity of ~85.33%.

SVM with radial kernel - The "True" dataset displays improved accuracy (=87.5%), specificity (~92.7%),
and also in sensitivity (72%) compared with the initial dataset.

SVM with polynomial kernel - The "True" dataset displays improved accuracy (=86.5%), sensitivity (=68%),
and a slight reduction in specificity (~92.7%) compared with the initial dataset.

K-nearest neighbors - The "True" dataset displays improved accuracy (=87.5%), sensitivity (=72%),
and also in specificity (~92.7%) compared with the initial dataset.

In all cases, there was an improvement in the accuracy and in at least one of the sensitivity or specificity
for the "True" dataset. However, with the improvements, the best model to use is very close between QDA and
random forests, although QDA is still just slightly better on both sensitivity and specificity than random forests.