Predictive Analytics: practical 3 solutions

The `OJ` data set

The `OJ` data set from the `ISLR` package contains information on which of two brands of orange juice customers purchased[1] and can be loaded using

[1] The response variable is `Purchase`.

```
data(OJ, package = "ISLR")
```

After loading the `caret` and `jrPredictive` package

```
library("caret")
library("jrPredictive")
```

make an initial examination of the relationships between each of the predictors and the response[2]

[2] Use the `plot` function with a model formula or the `pairs` function.

```
par(mfrow = c(4, 5), mar= c(4, .5, .5, .5))
plot(Purchase ~ ., data = OJ)
```

Initial model building

- To begin, create a logistic regression model that takes into consideration the prices of the two brands of orange juice, `PriceCH` and `PriceMM`.[3]

[3] Hint: Use the `train` function, with `method = 'glm'`. Look at the help page for the data set to understand what these variables represent.

```
m1 = train(Purchase ~ PriceCH + PriceMM,
    data = OJ, method = "glm")
```

- What proportion of purchases does this model get right?

```
mean(predict(m1) != OJ$Purchase)

## [1] 0.3776
```

- How does this compare to if we used no model?

```
# with no model we essentially predict according to
# proportion of observations in data
probs = table(OJ$Purchase)/nrow(OJ)
preds = sample(levels(OJ$Purchase), prob = probs)
mean(preds != OJ$Purchase)

## [1] 0.4991
```

Visualising the boundary

The `jrPredictive` package contains following code produces a plot of the decision boundary as seen in figure 1.

```
boundary_plot(m1,OJ$PriceCH, OJ$PriceMM, OJ$Purchase,
              xlab="Price CH", ylab="Price MM")
```

Run the boundary code above, and make sure you get a similar plot.

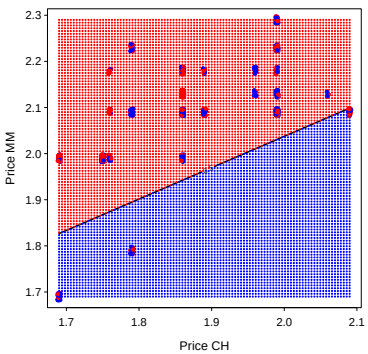

Figure 1: Examining the decision boundary for orange juice brand pur-

- What happens if we add an interaction term? How does the boundary change?

```
# We now have a curved decision boundary.
# There are two regions of where we would predict MM, bottom left, and a tiny one up in the top right.
```

- Try adding polynomial terms.

Using all of the predictors

- Fit a logistic regression model using all of the predictors.

```
mLM = train(Purchase ~ ., data = OJ, method = "glm")
```

- Is there a problem?

```
## YES!
```

We can view the most recent warning messages by using the `warnings` function

```
warnings()
```

This suggests some rank–deficient fit problems,

- Look at the final model, you should notice that a number of parameters have not been estimated

```
m_log$finalModel

##
## Call:  NULL
##
## Coefficients:
##    (Intercept)  WeekofPurchase         StoreID
##         5.1581         -0.0118         -0.1709
##        PriceCH         PriceMM          DiscCH
##         4.5865         -3.6249         10.7967
##         DiscMM       SpecialCH       SpecialMM
##        26.4615          0.2672          0.3169
##        LoyalCH     SalePriceMM     SalePriceCH
##        -6.3023              NA              NA
##      PriceDiff       Store7Yes       PctDiscMM
##             NA          0.3113        -50.6976
##      PctDiscCH   ListPriceDiff           STORE
##       -27.3399              NA              NA
##
## Degrees of Freedom: 1069 Total (i.e. Null);  1057 Residual
## Null Deviance:     1430
## Residual Deviance: 817  AIC: 843
```

The help page

```
?ISLR::OJ
```

gives further insight: the `PriceDiff` variable is a linear combination of `SalePriceMM` and `SalePriceCH` so we should remove this. In addition the `StoreID` and `STORE` variable are different encodings of the same information so we should remove one of these too. We also have `DiscCH` and `DiscMM` which are the differences between `PriceCH` and `SalePriceCH` and `PriceMM` and `SalePriceMM` respectively and `ListPriceDiff` is a linear combination of these prices. Removing all of these variables allows the model to be fit and all parameters to be estimated.[4]

[4] This is to highlight that we need to understand what we have in our data.

```
OJsub = OJ[,!(colnames(OJ) %in% c("STORE", "SalePriceCH",
          "SalePriceMM","PriceDiff", "ListPriceDiff"))]
OJsub$Store7 = as.numeric(OJsub$Store7) - 1
m_log = train(Purchase ~ ., data = OJsub, method = "glm")
```

The problem of linear combinations of predictors can be shown with this simple theoretical example. Suppose we have a response y and three predictors x_1, x_2 and the linear combination $x_3 = (x_1 + x_2)$. On fitting a linear model we try to find estimates of the parameters in the equation

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 (x_1 + x_2).$$

However we could just as easily rewrite this as

$$\begin{aligned} y &= \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 (x_1 + x_2) \\ &= \beta_0 + (\beta_1 + \beta_3) x_1 + (\beta_2 + \beta_3) x_2 \\ &= \beta_0 + \beta_1^* x_1 + \beta_2^* x_2. \end{aligned}$$

This leads to a rank–deficient model matrix, essentially we can never find the value of the β_3 due to the fact we have the linear combination of predictors.

We could achieve the same using the `caret` package function `findLinearCombos`. The function takes a model matrix as an argument. We can create such a matrix using the `model.matrix` function with our formula object

```
remove = findLinearCombos(
              model.matrix(Purchase ~ ., data = OJ))
```

The output list has a component called `remove` suggesting which variables should be removed to get rid of linear combinations

```
(badvar = colnames(OJ)[remove$remove])

## [1] "SalePriceMM"   "SalePriceCH"   "PriceDiff"
## [4] "ListPriceDiff" "STORE"

OJsub = OJ[, -remove$remove]
```

- How accurate is this new model using more predictors?]

```
# the corrected model
remove = findLinearCombos(model.matrix(Purchase~., data = OJ))
(badvar = colnames(OJ)[remove$remove])

## [1] "SalePriceMM"   "SalePriceCH"   "PriceDiff"
## [4] "ListPriceDiff" "STORE"

OJsub = OJ[,-(remove$remove)]
mLM = train(Purchase~., data = OJsub, method = "glm")
mean(predict(mLM,OJsub) == OJsub$Purchase)

## [1] 0.8355
```

- What are the values of sensitivity and specificity?

```
## could use confusionMatrix
(cmLM = confusionMatrix(predict(mLM,OJsub),OJsub$Purchase))

## Confusion Matrix and Statistics
##
##           Reference
## Prediction  CH  MM
##         CH 577 100
##         MM  76 317
##
##                Accuracy : 0.836
##                  95% CI : (0.812, 0.857)
##     No Information Rate : 0.61
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.651
##  Mcnemar's Test P-Value : 0.083
##
##             Sensitivity : 0.884
##             Specificity : 0.760
##          Pos Pred Value : 0.852
##          Neg Pred Value : 0.807
##              Prevalence : 0.610
##          Detection Rate : 0.539
##    Detection Prevalence : 0.633
##       Balanced Accuracy : 0.822
##
##        'Positive' Class : CH
##

# or
sensitivity(predict(mLM,OJsub),OJsub$Purchase)

## [1] 0.8836

specificity(predict(mLM,OJsub),OJsub$Purchase)

## [1] 0.7602
```

- What does this mean?

```
#The model is fairly good at picking up both positive events, person buys CH, and negative events, MM.
```

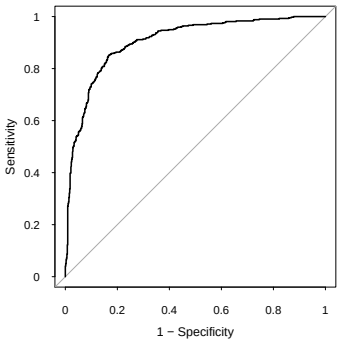

Figure 2: An example of a ROC curve for the logistic regression classifier. We can overlay ROC curves by adding the `add = TRUE` argument.

ROC curves

If we were interested in the area under the ROC curve, we could retrain the model using the `twoClassSummary` function as an argument to a train control object. Alternatively we can use the `pROC` package

```
library("pROC")
```

This also allows us to view the ROC curve, via

```
curve = roc(response = OJsub$Purchase,
  predictor = predict(m_log, type = "prob")[,"CH"])
## this makes CH the event of interest
plot(curve, legacy.axes = TRUE)
```

Other classification models

- Try fitting models using the other classification algorithms we have seen so far. To begin with, just have two covariates and use the `boundary_plot` function to visualise the results

We have seen LDA, QDA, KNN and logistic regression. Tomorrow we will cover support vector machines and neural nets; we can visualise the results in the same way.

```
mKNN = train(Purchase~., data = OJsub, method = "knn")
mLDA = train(Purchase~., data = OJsub, method = "lda")
mQDA = train(Purchase~., data = OJsub, method = "qda")
cmKNN = confusionMatrix(predict(mKNN,OJsub),OJsub$Purchase)
cmLDA = confusionMatrix(predict(mLDA,OJsub),OJsub$Purchase)
cmQDA = confusionMatrix(predict(mQDA,OJsub),OJsub$Purchase)
(info = data.frame(Model = c("logistic","knn","lda","qda"),
           Accuracy = c(cmLM$overall["Accuracy"],
               cmKNN$overall["Accuracy"],
               cmLDA$overall["Accuracy"],
               cmQDA$overall["Accuracy"]),
           Sensitivity = c(cmLM$byClass["Sensitivity"],
               cmKNN$byClass["Sensitivity"],
               cmLDA$byClass["Sensitivity"],
               cmQDA$byClass["Sensitivity"]),
           Specificity = c(cmLM$byClass["Specificity"],
               cmKNN$byClass["Specificity"],
               cmLDA$byClass["Specificity"],
               cmQDA$byClass["Specificity"])))
```

```
##      Model Accuracy Sensitivity Specificity
## 1 logistic   0.8355      0.8836      0.7602
## 2      knn   0.8065      0.8943      0.6691
## 3      lda   0.8374      0.8790      0.7722
## 4      qda   0.8168      0.8407      0.7794
```

- How do they compare?

```
#Logistic regression and LDA have highest accuracy, QDA is poor-
est at classifying events, KNN gives most false positives
```

- How does varying the number of nearest neighbours in a KNN affect the model fit?

```
#Accuracy increases at first with knn before then getting worse af-
ter a peak value of 9.
(mKNN2 = train(Purchase~., data = OJsub, method = "knn",
    tuneGrid = data.frame(k = 1:30)))

## k-Nearest Neighbors
##
## 1070 samples
##   12 predictors
##    2 classes: 'CH', 'MM'
##
## No pre-processing
## Resampling: Bootstrapped (25 reps)
## Summary of sample sizes: 1070, 1070, 1070, 1070, 1070, 1070, ...
## Resampling results across tuning parameters:
##
##   k   Accuracy  Kappa
##    1  0.6936    0.3559
##    2  0.6784    0.3245
##    3  0.6843    0.3339
##    4  0.6903    0.3438
##    5  0.6918    0.3443
##    6  0.6995    0.3596
##    7  0.7060    0.3725
##    8  0.7066    0.3726
##    9  0.7033    0.3646
##   10  0.7017    0.3605
##   11  0.6980    0.3519
##   12  0.6995    0.3548
##   13  0.6942    0.3414
##   14  0.6881    0.3276
##   15  0.6865    0.3236
##   16  0.6822    0.3146
##   17  0.6808    0.3112
##   18  0.6782    0.3050
##   19  0.6794    0.3083
##   20  0.6763    0.3011
##   21  0.6749    0.2968
##   22  0.6757    0.2981
##   23  0.6764    0.3002
##   24  0.6755    0.2973
##   25  0.6728    0.2916
##   26  0.6724    0.2904
##   27  0.6732    0.2923
##   28  0.6729    0.2925
##   29  0.6707    0.2869
##   30  0.6661    0.2774
##
## Accuracy was used to select the optimal model using
##   the largest value.
## The final value used for the model was k = 8.
```

The KNN algorithm described in the notes can also be used for regression problems. In this case the predicted response is the mean of the k nearest neighbours.

- Try fitting the KNN model for the regression problem in practical 1.

```
library("jrPredictive")
data(FuelEconomy, package = "AppliedPredictiveModeling")
regKNN = train(FE~., data = cars2010, method = "knn")
regLM = train(FE~., data = cars2010, method = "lm")
regKNN= validate(regKNN)
regLM = validate(regLM)
mark(regKNN)
mark(regLM)
```

- How does this compare to the linear regression models?

```
#The KNN regression model is not as good as the linear model at pre-
dicting the test set. It overestimates more at the lower end.
```

Resampling methods

- Fit a KNN regression model to the `cars2010` data set with `FE` as the response.

The data set can be loaded `data("FuelEconomy", package = "AppliedPredictiveModeling")`.

```
mKNN = train(FE ~ ., method = "knn", data = cars2010)
```

- Estimate test error using the validation set approach explored at the beginning of the chapter

```
# create a random sample to hold out
i = sample(nrow(cars2010), 100)
# set the train control object
tc = trainControl(method = "cv", number = 1,
    index = list(Fold1 = (1:nrow(cars2010))[-i]))
# fit the model using this train control object
mKNNvs = train(FE~., method = "knn", data = cars2010,
    trControl = tc)
```

- Using the same validation set, estimate the performance of the k nearest neighbours algorithm for different values of k.

```
mKNNvs2 = train(FE~., method = "knn", data = cars2010,
     trControl = tc, tuneGrid = data.frame(k= 2:20))
```

- Which model is chosen as the best when using the validation set approach?

```
## With set.seed(1)
mKNNvs2$bestTune

##   k
## 6 7
```

- Create new `trainControl` objects to specify the use of 5 fold and 10 fold cross validation as well as bootstrapping to estimate test MSE.

```
tc5fold = trainControl(method = "cv", number = 5)
tc10fold = trainControl(method = "cv", number = 10)
# use 50 boot strap estimates
tcboot = trainControl(method = "boot", number = 50)
```

- Go through the same training procedure attempting to find the best KNN model.

```
mKNNcv5 = train(FE~., data = cars2010, method = "knn",
    trControl = tc5fold, tuneGrid = data.frame(k = 2:20))

mKNNcv10 = train(FE~., data = cars2010, method = "knn",
    trControl = tc10fold, tuneGrid = data.frame(k = 2:20))

mKNNboot = train(FE~., data = cars2010, method = "knn",
    trControl = tcboot, tuneGrid = data.frame(k = 2:20))
mKNNcv5$bestTune

##   k
## 1 2

mKNNcv10$bestTune

##   k
## 1 2

mKNNboot$bestTune

##   k
## 1 2
```

- How do the results vary based on the method of estimation?

```
#The k-fold cross validation estimates and bootstrap estimates all
#yield the same conclusion, however it is different to when we used
#validation set approach earlier. We could plot the results
# from each on one plot to compare further:
plot(2:20, mKNNboot$results[,2], type = "l", ylab = "RMSE",
     xlab = "k", ylim = c(3,6.5))
lines(2:20, mKNNcv10$results[,2], col = "red")
lines(2:20, mKNNcv5$results[,2], col = "blue")
lines(2:20, mKNNvs2$results[,2], col = "green")
```

- Are the conclusions always the same?

```
#no see previous answer
```

If we add the `returnResamp = "all"` argument in the trainControl function we can plot the resampling distributions, see figure 3.

```
tc = trainControl(method = "cv", number = 15,
                  returnResamp = "all")
```

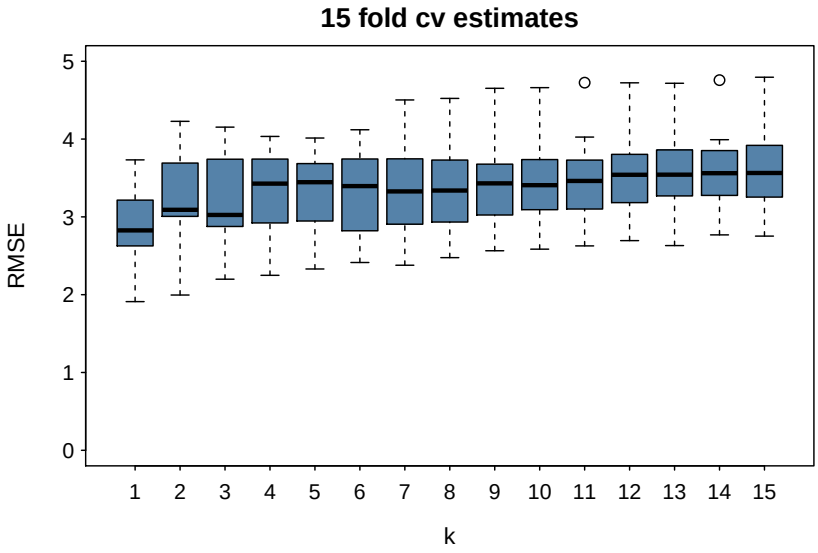

Figure 3: 15 fold cross validation estimates of RMSE in a K nearest neighbours model against number of nearest neighbours.

```
m = train(FE~., data = cars2010, method = "knn",
          tuneGrid = data.frame(k = 1:15), trControl = tc)
boxplot(RMSE~k, data = m$resample)
```

We can overlay the information from each method using `add = TRUE`. In addition we could compare the computational cost of each of the methods. The output list from a `train` object contains timing information which can be accessed

```
m$time
```

- Which method is the most computationally efficient?

```
mKNNvs2$time$everything

##    user  system elapsed
##   0.592   0.000   0.594

mKNNcv5$time$everything

##    user  system elapsed
##   1.772   0.000   1.773

mKNNcv10$time$everything

##    user  system elapsed
##   2.404   0.000   2.402

mKNNboot$time$everything

##    user  system elapsed
##   28.98    0.00   28.99
```

```
#The validation set approach was quickest, however we must bear in mind that the con-
clusion here
#was different to the other cross validation approaches. The two k-
-fold cross validation estimates of RMSE and the bootstrap
#estimates all agreed with each other lending more weight to their con-
clusions. Plus we saw in the lectures that validation set
#approach was prone to highly variable estimates meaning we could get a dif-
ferent conclusion using a different hold out set.
#Either of the two k--fold cross validation methods would be prefer-
able here.
```

An example with more than two classes

The `Glass` data set in the `mlbench` package is a data frame containing examples of the chemical analysis of 7 different types of glass. The goal is to be able to predict which category glass falls into based on the values of the 9 predictors.

```
data(Glass, package = "mlbench")
```

A logistic regression model is typically not suitable for more than 2 classes, so try fitting the other models using a training set that consists of 90% of the available data.

The function `createDataPartition` can be used here, see notes for a reminder.

Advanced

This section is intended for users who have a more in depth background to R programming. Attendance to the Programming in R course should be adequate background.

So far we have only used default functions and metrics to compare the performance of models, however we are not restricted to doing this. For example, training of classification models is typically more difficult when there is an imbalance in the two classes in the training set. Models trained from such data typically have high specificity but poor sensitivity or vice versa. Instead of training to maximise accuracy using data from the training set we could try to maximise according to some other criteria, namely sensitivity and specificity being as close to perfect as possible $(1, 1)$.

To add our function we need to make sure we mirror the structure of those included in caret already. The following code creates a new function that could be used to summarise a model

We can view a functions code by typing its name with no brackets.

```
fourStats = function (data, lev = NULL, model = NULL) {
  # This code will use the area under the ROC curve and the
  # sensitivity and specificity values from the built in
  # twoClassSummary function
  out = twoClassSummary(data, lev = levels(data$obs),
                        model = NULL)
  # The best possible model has sensitivity of 1 and
  # specifity of 1. How far are we from that value?
  coords = matrix(c(1, 1, out["Spec"], out["Sens"]),
                  ncol = 2,
                  byrow = TRUE)
  # return the disctance measure together with the
  # output from two class summary
  c(Dist = dist(coords)[1], out)
}
```

we could then use this in the `train` function

```
data(Sonar, package = "mlbench")
mod = train(Class ~ ., data = Sonar,
               method = "knn",
               # Minimize the distance to the perfect model
               metric = "Dist",
               maximize = FALSE,
               tuneLength = 20,
               trControl =
    trainControl(method = "cv", classProbs = TRUE,
                     summaryFunction = fourStats))
```

The `plot` function

```
plot(mod)
```

will then show the profile of the resampling estimates of our chosen statistic against the tuning parameters, see figure 4.

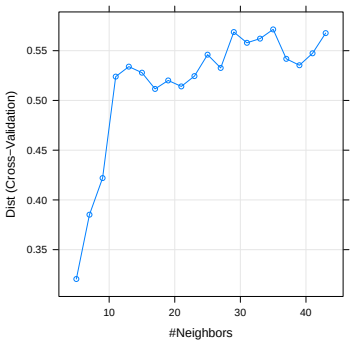

Figure 4: Plot of the distance from a perfect classifier measured by sensitivity and specificity against tuning parameter for a k nearest neighbour model.

- Have a go at writing a function that will allow a regression model to be chosen by the absolute value of the largest residual and try using it to fit a couple of models.

```
maxabsres = function(data, lev = NULL, model = NULL){
    m = max(abs(data$obs - data$pred))
    return(c("Max" = m))
}
# Test with pls regression
tccustom = trainControl(method = "cv",
                        summaryFunction = maxabsres)
mPLScustom = train(FE~., data = cars2010,
                   method = "pls",
                tuneGrid = data.frame(ncomp = 1:6),
                trControl = tccustom,
                metric = "Max", maximize = FALSE)
# success
# not to suggest this is a good choice of metric
```