

Practical 3 Solutions

Jumping Rivers

Predictive Analytics: practical 3

The OJ data set

The OJ data set from the ISLR package contains information on which of two brands of orange juice customers purchased¹ and can be loaded using

```
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 response²

```
par(mfrow = c(4, 5), mar = c(4, 0.5, 0.5, 0.5))
plot(Purchase ~ ., data = OJ)
```

Initial model building using logistic regression

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

```
getTrainPerf(m1)
```

```
##   TrainAccuracy TrainKappa method
## 1      0.6248998 0.08452175    glm
```

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

```
# with no model we essentially predict according to
# proportion of observations in data
```

```
# work out proportions
probs = table(OJ$Purchase)/nrow(OJ)
# sample using proportions
preds = sample(levels(OJ$Purchase), prob = probs)
# work out correct proportion
mean(preds != OJ$Purchase)
```

```
## [1] 0.5009346
```

¹The response variable is `Purchase`.

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

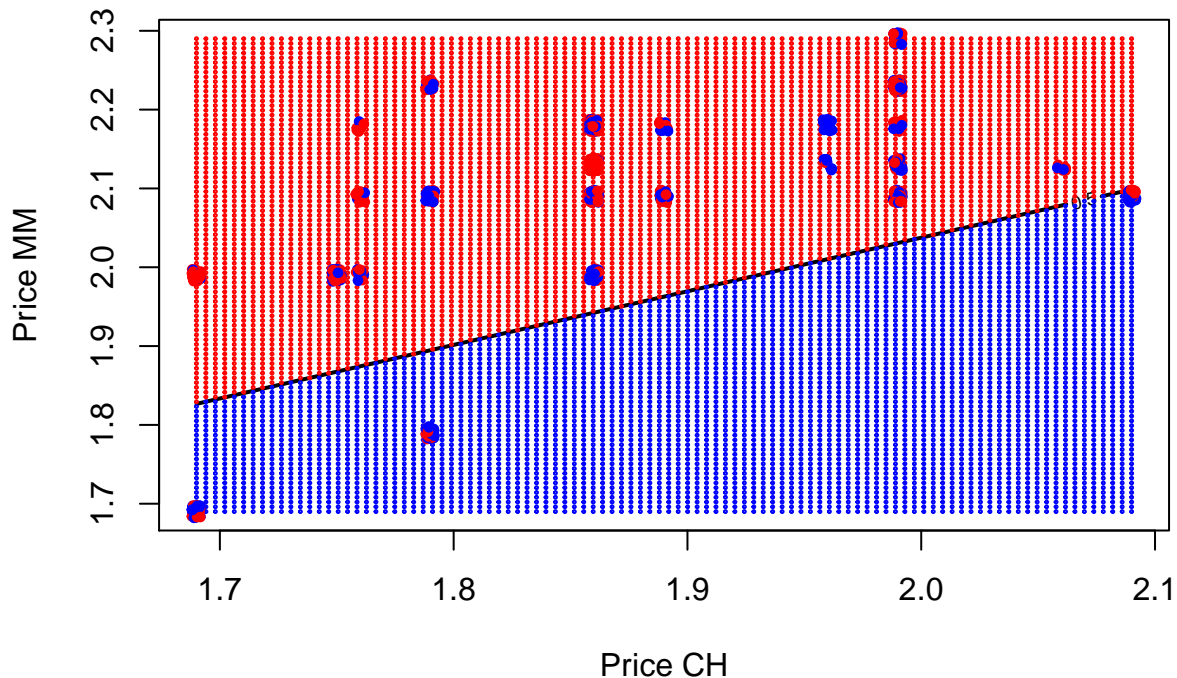


Figure 1: Examining the decision boundary for orange juice brand purchases by price.

- Use your model to predict if a customer will buy CH or MM if the price of CH and MM is 2.3 and 2.4 respectively

```
predict(m1, newdata = data.frame(PriceCH = 2.3, PriceMM = 2.4))
```

```
## [1] CH
## Levels: CH MM
```

Visualising the boundary

The `jrPred` 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.

- 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

- Instead of just using 2 predictors we want to use all of them. However, we have a few problems to tackle first. A few of our predictors are linear combinations of the others. This leads to what is called rank-deficiency problems. For instance, if you run the following model you'll realise there are a few NAs.

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

Take the predictor PriceDiff. It is impossible to estimate it's coefficient as it is a linear combination of PriceCH and PriceMM i.e. $\text{PriceDiff} = \text{PriceCH} - \text{PriceMM}$. In this particularly data set, there are quite a few linear combinations. We can find them using the `findLinearCombos()` and `model.matrix()` functions

```
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" "ListPriceDiff"
## [5] "STORE"
```

We can then remove these variable from the data

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

- Use the new OJsub data set to model Purchase using all of the predictors. How accurate is the model?

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

```
## TrainAccuracy TrainKappa method
## 1 0.8269648 0.6336684 glm
```

- 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 : 0.8355
##           95% CI : (0.8119, 0.8572)
##           No Information Rate : 0.6103
##           P-Value [Acc > NIR] : < 2e-16
##
##           Kappa : 0.6506
##           Mcnemar's Test P-Value : 0.08297
##
##           Sensitivity : 0.8836
##           Specificity : 0.7602
##           Pos Pred Value : 0.8523
##           Neg Pred Value : 0.8066
##           Prevalence : 0.6103
##           Detection Rate : 0.5393
##           Detection Prevalence : 0.6327
##           Balanced Accuracy : 0.8219
##
##           'Positive' Class : CH
##
```

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

```
## [1] 0.8836141
```

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

```
## [1] 0.7601918
```

- What does this mean?

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

K nearest neighbours

- Try fitting models using the K nearest neighbours algorithm. To begin with, just have two covariates and use the `boundary_plot` function to visualise the results.

```
mKNN = train(Purchase~., data = OJsub, method = "knn")
```

- How do they compare in accuracy, sensitivity and specificity?

```
cmKNN = confusionMatrix(predict(mKNN,OJsub),OJsub$Purchase)
(info = data.frame(Model = c("logistic","knn"),
  Accuracy = c(cmLM$overall["Accuracy"],
    cmKNN$overall["Accuracy"]),
  Sensitivity = c(cmLM$byClass["Sensitivity"],
    cmKNN$byClass["Sensitivity"]),
  Specificity = c(cmLM$byClass["Specificity"],
    cmKNN$byClass["Specificity"])))
```

```
##      Model  Accuracy Sensitivity Specificity
## 1 logistic 0.8355140   0.8836141   0.7601918
## 2      knn 0.8056075   0.8928025   0.6690647
```

- 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 after a peak value of 9.

```
(mKNN2 = train(Purchase~., data = OJsub, method = "knn",
  tuneGrid = data.frame(k = 1:30)))
```

```
## k-Nearest Neighbors
##
## 1070 samples
## 12 predictor
## 2 classes: 'CH', 'MM'
##
## No pre-processing
## Resampling: Bootstrapped (25 reps)
## Summary of sample sizes: 1070, 1070, 1070, 1070, 1070, ...
## Resampling results across tuning parameters:
##
##  k  Accuracy  Kappa
##  1  0.6856305  0.3420560
##  2  0.6684076  0.3061668
##  3  0.6795857  0.3283987
##  4  0.6898070  0.3476563
```

```
##      5  0.6964957  0.3599086
##      6  0.6992291  0.3661786
##      7  0.7044147  0.3738535
##      8  0.7009241  0.3662562
##      9  0.7034247  0.3697159
##     10  0.7059970  0.3756084
##     11  0.7032747  0.3685833
##     12  0.6963568  0.3519562
##     13  0.6916654  0.3422521
##     14  0.6926814  0.3429316
##     15  0.6861050  0.3273588
##     16  0.6889830  0.3340723
##     17  0.6890408  0.3339255
##     18  0.6858698  0.3268958
##     19  0.6825561  0.3189567
##     20  0.6791300  0.3099838
##     21  0.6780254  0.3076293
##     22  0.6791896  0.3102557
##     23  0.6795911  0.3107131
##     24  0.6787359  0.3082915
##     25  0.6749343  0.2991063
##     26  0.6747064  0.2993245
##     27  0.6762363  0.3016734
##     28  0.6758545  0.3016024
##     29  0.6738777  0.2961413
##     30  0.6754387  0.2994203
##
## Accuracy was used to select the optimal model using the largest value.
## The final value used for the model was k = 10.
```

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")
getTrainPerf(regKNN)
```

```
##      TrainRMSE TrainRsquared TrainMAE method
## 1  3.575063      0.7813084 2.443411      knn
```

```
getTrainPerf(regLM)
```

```
##      TrainRMSE TrainRsquared TrainMAE method
## 1  3.547022      0.7823438 2.460034      lm
```

- How does this compare to the linear regression models?

```
# The KNN regression model is not as good as the linear model, only just
```

Resampling methods

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

```
data(FuelEconomy, package = "AppliedPredictiveModeling")
```

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

- Estimate test error using 10-fold cross validation

```
# set the train control object
tc10fold = trainControl(method = "cv", number = 10)
# fit the model using this train control object
mKNN10 = train(FE ~ ., method = "knn", data = cars2010,
               trControl = tc10fold)
getTrainPerf(mKNN10)
```

```
##   TrainRMSE TrainRsquared TrainMAE method
## 1   3.307017      0.8099274  2.28837   knn
```

- Again using 10 fold CV, estimate the performance of the k nearest neighbours algorithm for different values of k .

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

- Which model is chosen as the best?

```
mKNNcv10$bestTune
```

```
##    k
## 2  3
```

- Create new `trainControl` objects to specify the use of 5 fold and 15 fold cross validation to estimate test RMSE.

```
tc5fold = trainControl(method = "cv", number = 5)
tc15fold = trainControl(method = "cv", number = 15)
```

- 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))

mKNNcv15 = train(FE ~ ., data = cars2010, method = "knn",
                 trControl = tc15fold, tuneGrid = data.frame(k = 2:20))
mKNNcv5$bestTune
```

```
##    k
## 1  2
```

```
mKNNcv15$bestTune
```

```
##    k
## 1  2
```

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 a k nearest neighbour model. Use k-fold cross validation if you want to. What proportion of predictions does your model get correct?

```
tc = trainControl(method = "cv", number = 10)
model = train(Type ~ ., data = Glass, trControl = tc, method = "knn")
getTrainPerf(model)
```

```
##      TrainAccuracy TrainKappa method
## 1      0.6792057   0.5575945     knn
```