Predictive Analytics: practical 3

```r
library("caret")
data(FuelEconomy, package = "AppliedPredictiveModeling")
set.seed(1)
```

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")`.

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

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

```r
# 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*.

```r
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?

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

##   k
## 2 3
```

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

```r
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 1.

```
tc = trainControl(method = "cv", number = 15,
                  returnResamp = "all")
m = train(FE~., data = cars2010, method = "knn",
          tuneGrid = data.frame(k = 1:15), trControl = tc)

boxplot(RMSE~k, data = m$resample)
```

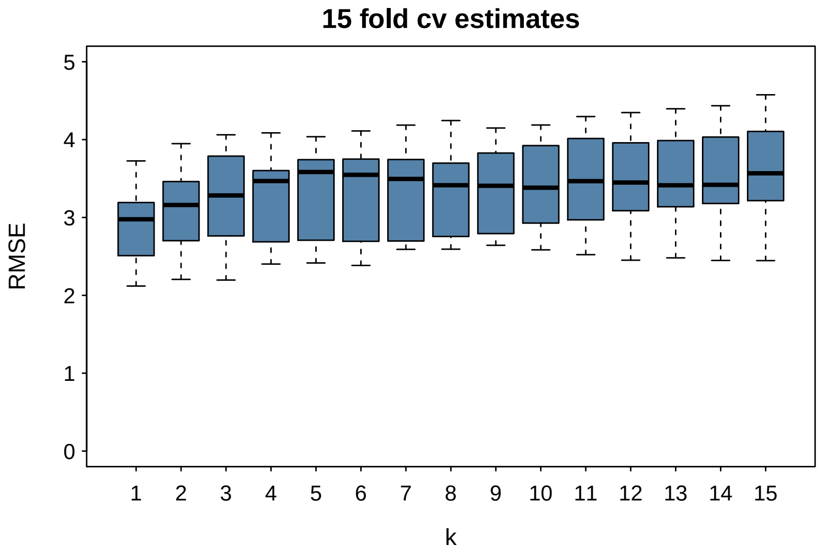

Figure 1: 15 fold cross validation estimates of RMSE in a K nearest neighbours model against number of nearest neighbours.

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
##   0.412   0.000   0.414

mKNNcv5$time$everything

##    user  system elapsed
##   1.484   0.000   1.490

mKNNcv10$time$everything

##    user  system elapsed
##   1.784   0.000   1.790

mKNNboot$time$everything

##    user  system elapsed
##   25.57    0.00   25.65

# The validation set approach was quickest, however we must bear in mind
# that the conclusion 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 conclusions. Plus we saw in the lectures that validation set
# approach was prone to highly variable estimates meaning we could get a
# different conclusion using a different hold out set. Either of the two
# k--fold cross validation methods would be preferable here.
```

Penalised regression

The `diabetes` data set in the `lars` package contains measurements of a number of predictors to model a response y, a measure of disease progression. There are other columns in the data set which contain interactions so we will extract just the predictors and the response. The data has already been normalized.

```
data(diabetes, package = "lars")
diabetesdata = cbind(diabetes$x,"y" = diabetes$y)
```

- Try fitting a lasso, ridge and elastic net model using all of the main effects, pairwise interactions and square terms from each of the predictors.[1]

```
## load the data in
modelformula = as.formula(paste("y~(.)^2 + ",
    paste0("I(",colnames(diabetesdata),"^2)",
          collapse = "+")
    ))
mLASSO = train(modelformula, data = diabetesdata,
    method = "lasso")
mRIDGE = train(modelformula, data = diabetesdata,
    method = "ridge")
mENET = train(modelformula, data = diabetesdata,
    method = "enet")
```

- Try to narrow in on the region of lowest RMSE for each model, don't forget about the `tuneGrid` argument to the train function.

```
# examine previous output then train over a finer grid near the better end
mLASSOfine = train(modelformula,data = diabetesdata,
    method = "lasso", tuneGrid = data.frame(fraction = seq(0.1,0.5,by = 0.05)))

## Warning:  model fit failed for Resample01:  fraction=0.5
Error in if (zmin < gamhat) { :  missing value where TRUE/FALSE
needed
## Warning in nominalTrainWorkflow(x = x, y = y, wts = weights,
info = trainInfo, :  There were missing values in resampled
performance measures.

mLASSOfine$results
```

fraction = 0 is the same as the null model.
$y \sim (.) \wedge 2$ is short hand for a model that includes pairwise interactions for each predictor, so if we use this we should only need to add the square terms

[1] Hint: see notes for shortcut on creating model formula. Also be aware that if the predictor is a factor a polynomial term doesn't make sense

```
##   fraction  RMSE Rsquared RMSESD RsquaredSD
## 1     0.10 17.24   0.9496  1.045   0.006322
## 2     0.15 17.62   0.9474  1.010   0.006045
## 3     0.20 17.84   0.9461  1.084   0.006533
## 4     0.25 17.97   0.9454  1.154   0.006872
## 5     0.30 18.06   0.9448  1.217   0.007202
## 6     0.35 18.12   0.9445  1.285   0.007623
## 7     0.40 18.16   0.9443  1.342   0.007992
## 8     0.45 18.19   0.9441  1.403   0.008400
## 9     0.50 18.23   0.9438  1.464   0.008831
```

```
# best still right down at the 0.1 end
mLASSOfiner = train(modelformula,data = diabetesdata,
    method = "lasso", tuneGrid = data.frame(fraction = seq(0.01,0.15,by = 0.01)))
```

```
## Warning:  model fit failed for Resample24:  fraction=0.15
Error in if (zmin < gamhat) { :  missing value where TRUE/FALSE
needed
## Warning in nominalTrainWorkflow(x = x, y = y, wts = weights,
info = trainInfo, :  There were missing values in resampled
performance measures.
```

```
mLASSOfiner$results
```

```
##    fraction  RMSE Rsquared RMSESD RsquaredSD
## 1      0.01 48.82   0.9543 16.879   0.002663
## 2      0.02 32.54   0.9543 17.893   0.003012
## 3      0.03 26.27   0.9548 15.299   0.003783
## 4      0.04 23.11   0.9540 12.375   0.004271
## 5      0.05 21.16   0.9533  9.923   0.005061
## 6      0.06 19.90   0.9530  7.726   0.005499
## 7      0.07 19.04   0.9523  5.709   0.005537
## 8      0.08 18.33   0.9516  4.045   0.005596
## 9      0.09 17.85   0.9513  2.701   0.006130
## 10     0.10 17.56   0.9508  1.682   0.006680
## 11     0.11 17.39   0.9502  1.289   0.006990
## 12     0.12 17.39   0.9496  1.312   0.007265
## 13     0.13 17.47   0.9491  1.378   0.007689
## 14     0.14 17.56   0.9486  1.426   0.008035
## 15     0.15 17.65   0.9480  1.467   0.008370
```

```
# 0.09 seems the best

mRIDGEfine = train(modelformula,data = diabetesdata,
    method = "ridge", tuneGrid = data.frame(lambda = seq(0,0.1,by = 0.01)))
mRIDGEfine$results
```

```
##    lambda  RMSE Rsquared RMSESD RsquaredSD
## 1    0.00 18.03   0.9459 1.0195   0.007486
## 2    0.01 16.90   0.9522 0.8943   0.005874
## 3    0.02 16.83   0.9525 0.8872   0.005968
## 4    0.03 16.87   0.9523 0.9050   0.006233
## 5    0.04 16.96   0.9517 0.9359   0.006585
## 6    0.05 17.10   0.9509 0.9737   0.006985
## 7    0.06 17.28   0.9498 1.0148   0.007411
## 8    0.07 17.48   0.9487 1.0574   0.007853
## 9    0.08 17.70   0.9474 1.1002   0.008303
## 10   0.09 17.94   0.9460 1.1426   0.008758
## 11   0.10 18.19   0.9445 1.1843   0.009214
```

```
mRIDGEfiner = train(modelformula,data = diabetesdata,
    method = "ridge", tuneGrid = data.frame(lambda = seq(0.005,0.03,by = 0.001)))
mRIDGEfiner$results
```

```
##    lambda  RMSE Rsquared RMSESD RsquaredSD
## 1   0.005 16.69   0.9525 0.7355   0.003735
## 2   0.006 16.66   0.9526 0.7287   0.003736
## 3   0.007 16.63   0.9528 0.7233   0.003747
## 4   0.008 16.61   0.9529 0.7189   0.003764
## 5   0.009 16.59   0.9530 0.7153   0.003787
## 6   0.010 16.58   0.9530 0.7124   0.003813
## 7   0.011 16.57   0.9531 0.7102   0.003841
## 8   0.012 16.56   0.9531 0.7085   0.003872
## 9   0.013 16.55   0.9532 0.7072   0.003904
## 10  0.014 16.54   0.9532 0.7063   0.003938
## 11  0.015 16.54   0.9532 0.7057   0.003972
## 12  0.016 16.54   0.9532 0.7055   0.004008
## 13  0.017 16.54   0.9532 0.7055   0.004044
## 14  0.018 16.54   0.9532 0.7058   0.004081
## 15  0.019 16.54   0.9532 0.7063   0.004118
## 16  0.020 16.54   0.9532 0.7070   0.004156
## 17  0.021 16.54   0.9532 0.7079   0.004195
## 18  0.022 16.55   0.9531 0.7090   0.004233
## 19  0.023 16.55   0.9531 0.7102   0.004272
## 20  0.024 16.56   0.9531 0.7115   0.004311
## 21  0.025 16.56   0.9530 0.7130   0.004351
## 22  0.026 16.57   0.9530 0.7146   0.004390
## 23  0.027 16.58   0.9529 0.7163   0.004430
## 24  0.028 16.59   0.9529 0.7182   0.004469
## 25  0.029 16.59   0.9528 0.7201   0.004509
## 26  0.030 16.60   0.9528 0.7221   0.004549
```

```
# 0.023 seems best

mENETfine = train(modelformula, data = diabetesdata,
    method = "enet", tuneGrid = expand.grid(
                         lambda = c(0.001,0.01,0.1),
                         fraction = c(0.4,0.5,0.6)
    ))
mENETfine$results
```

```
##   lambda fraction  RMSE Rsquared RMSESD RsquaredSD
## 1  0.001      0.4 16.37   0.9568 0.7865   0.003754
## 4  0.010      0.4 16.53   0.9579 1.1314   0.003818
## 7  0.100      0.4 22.85   0.9553 2.5052   0.003438
## 2  0.001      0.5 16.76   0.9547 0.8179   0.004496
## 5  0.010      0.5 15.93   0.9590 0.7400   0.003178
## 8  0.100      0.5 17.05   0.9566 1.1056   0.004016
## 3  0.001      0.6 16.95   0.9537 0.8400   0.004805
## 6  0.010      0.6 16.28   0.9573 0.7709   0.003461
## 9  0.100      0.6 16.72   0.9547 0.8511   0.004989

mENETfiner = train(modelformula, data = diabetesdata,
    method = "enet", tuneGrid = expand.grid(
                         lambda = seq(0.001,0.1,length.out = 10),
                         fraction = 0.5))
mENETfiner$results

##    lambda fraction  RMSE Rsquared RMSESD RsquaredSD
## 1   0.001      0.5 16.76   0.9539 0.8983   0.004779
## 2   0.012      0.5 15.94   0.9581 0.6089   0.003401
## 3   0.023      0.5 15.96   0.9583 0.6562   0.003541
## 4   0.034      0.5 16.16   0.9579 0.6988   0.003580
## 5   0.045      0.5 16.38   0.9574 0.7513   0.003529
## 6   0.056      0.5 16.57   0.9570 0.8163   0.003579
## 7   0.067      0.5 16.71   0.9567 0.8622   0.003605
## 8   0.078      0.5 16.82   0.9565 0.8949   0.003685
## 9   0.089      0.5 16.92   0.9562 0.9163   0.003809
## 10  0.100      0.5 16.99   0.9559 0.9337   0.003945

# 0.012, 0.5 best
```

We can view what the coefficients will be by using

```
coef = predict(m.lasso$finalModel,
        mode = "fraction",
        s = 0.1,# which ever fraction was chosen as best
        type = "coefficients"
)
```

- How many features have been chosen by the lasso and enet models?

```
# use predict to find the coefficients
coefLASSO = predict(mLASSOfiner$finalModel, mode = "fraction",
        type = "coefficient", s = 0.09
        )
sum(coefLASSO$coefficients != 0)

## [1] 54

coefENET= predict(mENETfiner$finalModel, mode = "fraction",
        type = "coefficient", s = 0.5
        )
sum(coefENET$coefficients != 0)

## [1] 21
```

- How do these models compare to principal components and partial least squares regression?

```
mPCR = train(modelformula, data = diabetesdata, method = "pcr", tuneGrid = data.frame(ncomp = 1:7))
mPLS = train(modelformula, data = diabetesdata, method = "pls", tuneGrid = data.frame(ncomp = 1:7))
mPLS2 = train(modelformula, data = diabetesdata, method = "pls", tuneGrid = data.frame(ncomp = 5:15))
getTrainPerf(mLASSOfiner)

##   TrainRMSE TrainRsquared method
## 1     17.39        0.9496  lasso

getTrainPerf(mRIDGEfiner)

##   TrainRMSE TrainRsquared method
## 1     16.54        0.9532  ridge

getTrainPerf(mENETfiner)

##   TrainRMSE TrainRsquared method
## 1     15.94        0.9581   enet

getTrainPerf(mPCR)

##   TrainRMSE TrainRsquared method
## 1     16.55        0.9557    pcr

getTrainPerf(mPLS2)

##   TrainRMSE TrainRsquared method
## 1     15.79        0.9585    pls

# The elastic net model has the lowest estimated test error, all are fairly
# similar. The elastic net model suggests only 21 non--zero coefficients out
# of all of those included in the model.
```

Advanced

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 $(1, 1)$.

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.

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 will then show the profile of the resampling estimates of our chosen statistic against the tuning parameters, see figure 2.

```
plot(mod)
```

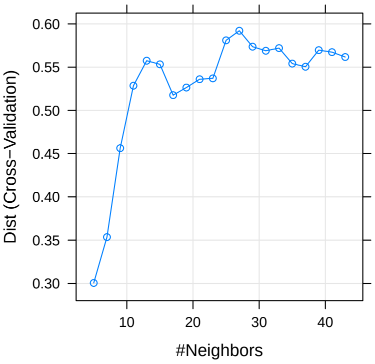

Figure 2: Plot of the distance from a perfect classifier measured by sensitivity and specificity against tuning parameter for a k nearest neighbour model.

- 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)
```

```r
mPLScustom = train(FE ~ ., data = cars2010, method = "pls", tuneGrid = data.frame(ncomp = 1:6),
    trControl = tccustom, metric = "Max", maximize = FALSE)
# success not to sugges this is a good choice of metric
```