Predictive Analytics: practical 2 solutions

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

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.

```r
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]

[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

```r
## 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")
```

`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

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

```r
# 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)))
mLASSOfine$results

##   fraction  RMSE Rsquared RMSESD RsquaredSD
## 1     0.10 17.48   0.9498  1.199   0.007721
## 2     0.15 17.88   0.9475  1.460   0.009881
## 3     0.20 18.19   0.9456  1.704   0.011844
## 4     0.25 18.36   0.9445  1.855   0.012976
## 5     0.30 18.50   0.9437  1.968   0.013817
## 6     0.35 18.58   0.9432  1.999   0.014093
## 7     0.40 18.62   0.9429  2.035   0.014408
## 8     0.45 18.66   0.9427  2.075   0.014785
## 9     0.50 18.69   0.9425  2.086   0.014922

# best still right down at the 0.1 end
mLASSOfiner = train(modelformula, data = diabetesdata, method = "lasso", tuneGrid = data.frame(fraction = seq(0.
    0.15, by = 0.01)))
mLASSOfiner$results

##    fraction  RMSE Rsquared  RMSESD RsquaredSD
## 1      0.01 39.67   0.9562 19.6313   0.005303
## 2      0.02 26.91   0.9549 15.2340   0.004879
## 3      0.03 22.03   0.9546 10.9004   0.005734
## 4      0.04 19.57   0.9534  7.7419   0.006264
## 5      0.05 18.36   0.9528  5.6974   0.006430
## 6      0.06 17.85   0.9520  4.1728   0.006489
## 7      0.07 17.61   0.9514  2.9337   0.006431
## 8      0.08 17.48   0.9508  1.8775   0.006491
## 9      0.09 17.39   0.9502  1.1800   0.006490
## 10     0.10 17.37   0.9498  0.9972   0.006489
## 11     0.11 17.41   0.9496  1.0084   0.006611
## 12     0.12 17.47   0.9493  1.0294   0.006780
## 13     0.13 17.53   0.9490  1.0602   0.006985
## 14     0.14 17.59   0.9487  1.0933   0.007181
## 15     0.15 17.65   0.9485  1.1266   0.007329
```

```
# 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 18.50   0.9437 2.2311   0.013700
## 2    0.01 17.08   0.9518 0.8079   0.004190
## 3    0.02 17.02   0.9519 0.8167   0.004613
## 4    0.03 17.07   0.9515 0.8462   0.005145
## 5    0.04 17.19   0.9508 0.8911   0.005730
## 6    0.05 17.35   0.9498 0.9457   0.006342
## 7    0.06 17.54   0.9486 1.0055   0.006962
## 8    0.07 17.77   0.9472 1.0674   0.007583
## 9    0.08 18.01   0.9457 1.1299   0.008199
## 10   0.09 18.27   0.9441 1.1920   0.008810
## 11   0.10 18.55   0.9425 1.2531   0.009412
```

```
mRIDGEfiner = train(modelformula, data = diabetesdata, method = "ridge", tuneGrid = data.frame(lambda = seq(0.00
    0.03, by = 0.001)))
mRIDGEfiner$results
```

```
##    lambda  RMSE Rsquared RMSESD RsquaredSD
## 1   0.005 17.05   0.9516 0.7883   0.004174
## 2   0.006 17.01   0.9518 0.7747   0.004126
## 3   0.007 16.99   0.9520 0.7639   0.004092
## 4   0.008 16.96   0.9521 0.7549   0.004067
## 5   0.009 16.94   0.9522 0.7472   0.004048
## 6   0.010 16.93   0.9522 0.7407   0.004035
## 7   0.011 16.92   0.9523 0.7351   0.004027
## 8   0.012 16.91   0.9523 0.7302   0.004022
## 9   0.013 16.90   0.9524 0.7261   0.004021
## 10  0.014 16.89   0.9524 0.7225   0.004023
## 11  0.015 16.89   0.9524 0.7195   0.004028
## 12  0.016 16.88   0.9524 0.7170   0.004035
## 13  0.017 16.88   0.9524 0.7149   0.004044
## 14  0.018 16.88   0.9524 0.7132   0.004056
## 15  0.019 16.88   0.9524 0.7119   0.004069
## 16  0.020 16.88   0.9524 0.7110   0.004084
## 17  0.021 16.88   0.9523 0.7104   0.004101
## 18  0.022 16.89   0.9523 0.7101   0.004119
## 19  0.023 16.89   0.9523 0.7100   0.004139
## 20  0.024 16.89   0.9523 0.7102   0.004160
## 21  0.025 16.90   0.9522 0.7107   0.004182
## 22  0.026 16.91   0.9522 0.7113   0.004205
## 23  0.027 16.91   0.9521 0.7122   0.004229
## 24  0.028 16.92   0.9521 0.7132   0.004254
## 25  0.029 16.93   0.9520 0.7144   0.004280
## 26  0.030 16.94   0.9520 0.7158   0.004306

# 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 16.24   0.9563 1.0730   0.004472
## 4  0.010      0.4 16.13   0.9586 1.0273   0.003310
## 7  0.100      0.4 22.74   0.9557 2.0464   0.003014
## 2  0.001      0.5 16.66   0.9539 1.1465   0.004998
## 5  0.010      0.5 15.77   0.9588 0.8909   0.003513
## 8  0.100      0.5 16.86   0.9568 0.8872   0.003665
## 3  0.001      0.6 16.93   0.9524 1.1989   0.005321
## 6  0.010      0.6 16.16   0.9566 0.9615   0.004049
## 9  0.100      0.6 16.59   0.9541 0.9030   0.005057

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 16.81   0.9536  1.085   0.005505
## 2   0.012      0.5 15.82   0.9590  1.035   0.004233
## 3   0.023      0.5 15.84   0.9592  1.035   0.004195
## 4   0.034      0.5 16.03   0.9585  1.027   0.004228
## 5   0.045      0.5 16.27   0.9578  1.037   0.004125
## 6   0.056      0.5 16.49   0.9574  1.064   0.004095
## 7   0.067      0.5 16.67   0.9570  1.099   0.004207
## 8   0.078      0.5 16.82   0.9568  1.131   0.004292
## 9   0.089      0.5 16.93   0.9566  1.159   0.004414
## 10  0.100      0.5 17.02   0.9563  1.167   0.004539

# 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     17.37        0.9498  lasso

getTrainPerf(mRIDGEfiner)

##   TrainRMSE TrainRsquared method
## 1     16.88        0.9524  ridge

getTrainPerf(mENETfiner)

##   TrainRMSE TrainRsquared method
## 1     15.82         0.959   enet

getTrainPerf(mPCR)

##   TrainRMSE TrainRsquared method
## 1     16.62        0.9553    pcr

getTrainPerf(mPLS2)

##   TrainRMSE TrainRsquared method
## 1     15.92        0.9575    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

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

function that could be used to summarise a model

```
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 1.

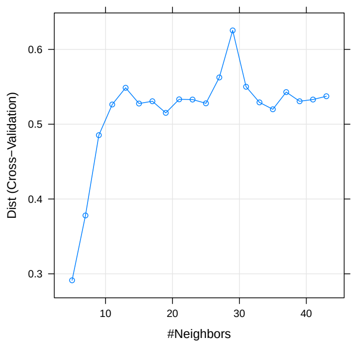

Figure 1: Plot of the distance from a perfect classifier measured by sensitivity and specificity against tuning parameter for a *k* nearest neighbour model.

- 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
```