IS624 Assignment3

James Quacinella

06/22/2015

Contents

Question 7.2

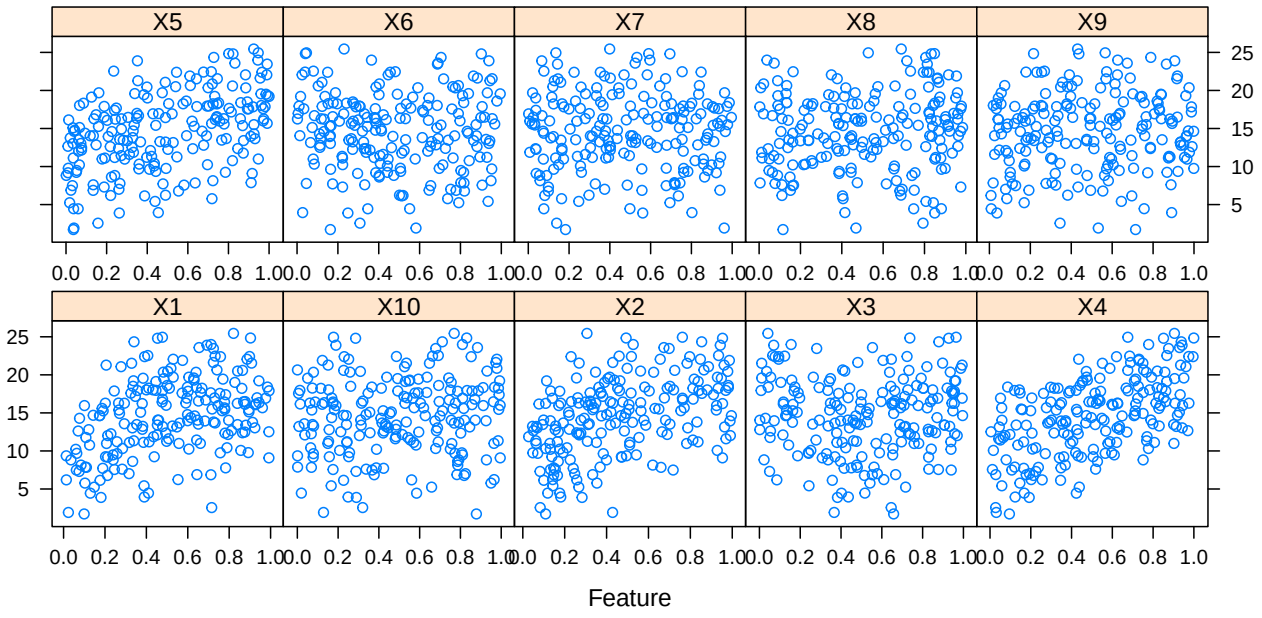
Question

Friedman (1991) introduced several benchmark data sets create by simulation. One of these simulations used the following nonlinear equation to create data:

$$y = 10sin(nx_1x_2) + 20(x_3 - 0.5)2 + 10x_4 + 5x_5 + N(0, alpha^2)$$

where the x values are random variables uniformly distributed between [0, 1] (there are also 5 other non-informative variables also created in the simula- tion). The package mlbench contains a function called mlbench.friedman1 that simulates these data:

```
trainingData <- mlbench.friedman1(200, sd = 1)
## We convert the ' x ' data from a matrix to a data
## frame One reason is that this will give the
## columns names.
trainingData$x <- data.frame(trainingData$x)
## Look at the data using
featurePlot(trainingData$x, trainingData$y)
```




```
## or other methods.  This creates a list with a
## vector ' y ' and a matrix of predictors ' x ' .
## Also simulate a large test set to estimate the
## true error rate with good precision:
testData <- mlbench.friedman1(5000, sd = 1)
testData$x <- data.frame(testData$x)
```

Tune several models on these data. For example:

```
knnModel <- train(x = trainingData$x, y = trainingData$y,
    method = "knn", preProc = c("center", "scale"),
    tuneLength = 10)

knnModel
```

```
## k-Nearest Neighbors
##
## 200 samples
##  10 predictors
##
## Pre-processing: centered, scaled
## Resampling: Bootstrapped (25 reps)
##
## Summary of sample sizes: 200, 200, 200, 200, 200, 200, ...
##
## Resampling results across tuning parameters:
##
##   k   RMSE      Rsquared   RMSE SD    Rsquared SD
##    5  3.492611  0.5514878  0.1890666  0.05654488
##    7  3.367499  0.5919769  0.1854113  0.05577892
##    9  3.309609  0.6182302  0.2022743  0.05494413
##   11  3.289578  0.6367648  0.2270144  0.04463323
##   13  3.281397  0.6559647  0.2285598  0.04659637
```

```
##   15  3.287363  0.6661678  0.2404266  0.04807383
##   17  3.334535  0.6641188  0.2509047  0.05246330
##   19  3.370126  0.6642655  0.2527395  0.05007358
##   21  3.403700  0.6655961  0.2556742  0.05093303
##   23  3.435312  0.6667585  0.2498849  0.05135945
##
## RMSE was used to select the optimal model using  the smallest value.
## The final value used for the model was k = 13.
```

```
knnPred <- predict(knnModel, newdata = testData$x)
## The function ' postResample ' can be used to get
## the test set perforamnce values
postResample(pred = knnPred, obs = testData$y)
```

```
##      RMSE  Rsquared
## 3.1454316 0.6661258
```

Answer

A kNN model was generated for us above. We have three options: train a neural network, a MARS model, and a SVM model (with potentially different kernels). Lets first look at a neural net model.

Neural Net Model

```
# Using train() led to worse results, and I am not
# sure what the difference is in using train versus
# nnet nnetGrid <- expand.grid(.decay = c(0.01),
# .size=c(5)) nnetFit <- train(trainingData$x,
# trainingData$y, method = 'nnet', linout = TRUE,
# trace = FALSE, maxit = 500, tuneGrid = nnetGrid,
# MaxNWts = 5 * (ncol(trainingData$x) + 1) + 5 + 1)

# Create a Neural Net model
nnetFit <- nnet(trainingData$x, trainingData$y, size = 5,
    decay = 0.01, linout = TRUE, trace = FALSE, maxit = 500,
    MaxNWts = 5 * (ncol(trainingData$x) + 1) + 5 +
        1)

# Show the fit
nnetFit
```

```
## a 10-5-1 network with 61 weights
## options were - linear output units  decay=0.01
```

```
summary(nnetFit)
```

```
## a 10-5-1 network with 61 weights
## options were - linear output units  decay=0.01
##   b->h1  i1->h1  i2->h1  i3->h1  i4->h1  i5->h1  i6->h1  i7->h1  i8->h1
##   -0.87   -0.97   -0.67    1.04    1.20    0.50    0.00    0.12    0.03
##  i9->h1 i10->h1
##    0.08   -0.15
##   b->h2  i1->h2  i2->h2  i3->h2  i4->h2  i5->h2  i6->h2  i7->h2  i8->h2
##    0.70   -0.94   -0.56   -6.14    0.33    0.16    0.41    0.16    0.42
##  i9->h2 i10->h2
##   -0.13    0.09
##   b->h3  i1->h3  i2->h3  i3->h3  i4->h3  i5->h3  i6->h3  i7->h3  i8->h3
##   -4.22    3.73   12.72    1.24   -0.99   -0.10    0.07   -0.69   -0.51
##  i9->h3 i10->h3
##   -0.19    0.40
##   b->h4  i1->h4  i2->h4  i3->h4  i4->h4  i5->h4  i6->h4  i7->h4  i8->h4
##    0.02   -7.52    0.88   -1.00    0.16   -0.03    0.23    0.26    0.74
##  i9->h4 i10->h4
##    0.42   -0.08
##   b->h5  i1->h5  i2->h5  i3->h5  i4->h5  i5->h5  i6->h5  i7->h5  i8->h5
##   -8.50    7.68    7.82   -0.09   -0.40    0.44   -0.31    0.12   -0.11
##  i9->h5 i10->h5
##    0.13    0.70
```

```
##   b->o  h1->o  h2->o  h3->o  h4->o  h5->o
## -12.65  37.16  18.40   6.72 -11.46   9.49
```

```
# Predict on the test set and see how well it did
nnetPredict <- predict(nnetFit, testData$x)
postResample(pred = nnetPredict, obs = testData$y)
```

```
##      RMSE  Rsquared
## 1.4423934 0.9167035
```

Discussion: This basic model ends up with a high R^2 with no tuning of any model parameters. However, I tried creating a model using tain() instead of nnet(), but the results seemed worse off for some reason. I am not sure what the difference is between train() and using the model directly.

MARS Model

```
# Create a MARS model
marsFit <- earth(trainingData$x, trainingData$y)

# Show the fit
marsFit
```

```
## Selected 12 of 20 terms, and 7 of 10 predictors
## Termination condition: Reached nk 21
## Importance: X4, X2, X1, X5, X3, X10, X8, X6-unused, X7-unused, ...
## Number of terms at each degree of interaction: 1 11 (additive model)
## GCV 2.802104    RSS 438.9356    GRSq 0.8927606    RSq 0.9151612
```

```
summary(marsFit)
```

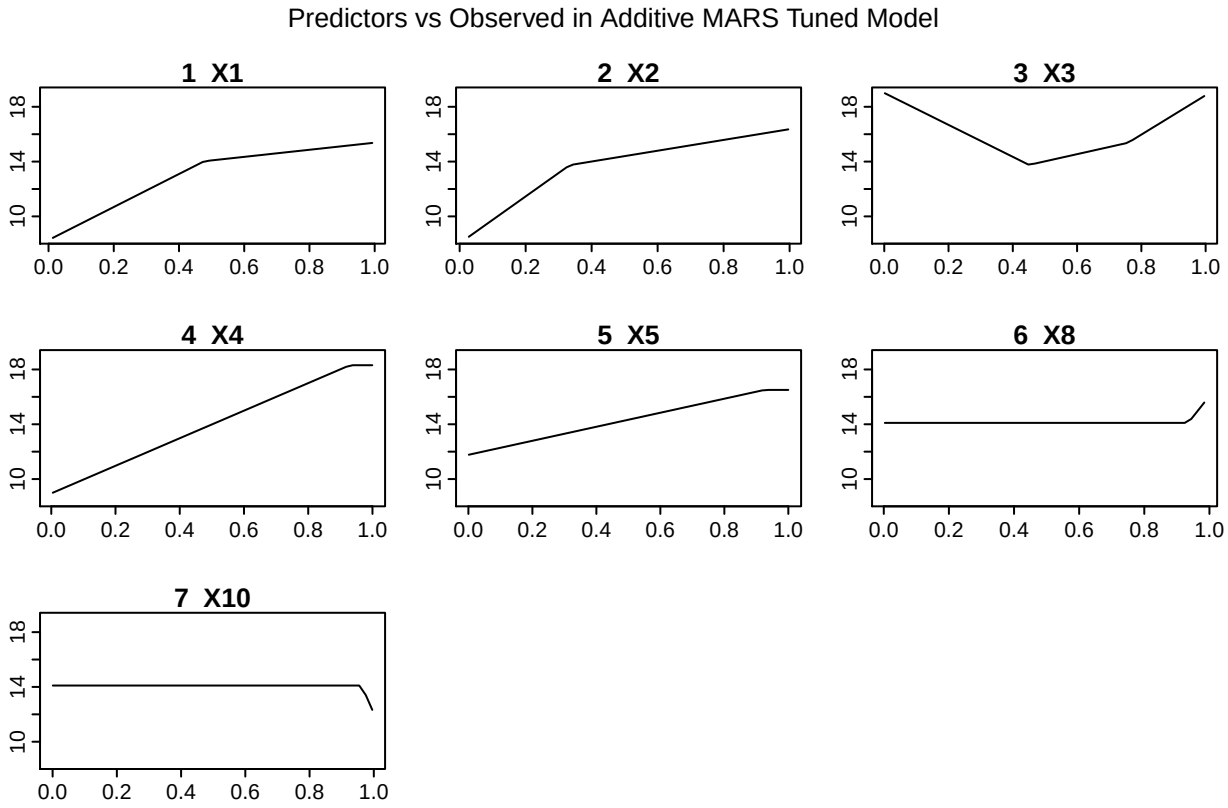
```
## Call: earth(x=trainingData$x, y=trainingData$y)
##
##                 coefficients
## (Intercept)        19.942097
## h(0.47762-X1)     -12.049125
## h(X1-0.47762)       2.584661
## h(0.333521-X2)    -17.108629
## h(X2-0.333521)      3.938851
## h(0.450485-X3)     11.661496
## h(X3-0.450485)      5.244239
## h(X3-0.758697)      9.131904
## h(0.929743-X4)    -10.060992
## h(0.925323-X5)     -5.119769
## h(X8-0.934716)     29.960793
## h(X10-0.961949)   -53.215152
##
## Selected 12 of 20 terms, and 7 of 10 predictors
## Termination condition: Reached nk 21
## Importance: X4, X2, X1, X5, X3, X10, X8, X6-unused, X7-unused, ...
## Number of terms at each degree of interaction: 1 11 (additive model)
## GCV 2.802104    RSS 438.9356    GRSq 0.8927606    RSq 0.9151612
```

```
# Predict on the test set and see how well it did
marsPredict <- predict(marsFit, testData$x)
postResample(pred = marsPredict, obs = testData$y)
```

```
##      RMSE  Rsquared
## 1.8871553 0.8598973
```

```
# Inspect the model's use of predictors
plotmo(marsFit, caption = "Predictors vs Observed in Additive MARS Tuned Model")
```

```
##  grid:    X1        X2        X3        X4      X5        X6        X7
##    0.5054907 0.4244831 0.5158712 0.5114502 0.45547 0.4441768 0.4696555
##         X8        X9       X10
##  0.4949538 0.4917092 0.5036008
```



Discussion: It does seem like the first 5 predictors are noted as important, and left the other predictors out. Pretty impressive there, MARS. Lets try what the book has for tuning this model:

```
marsGrid <- expand.grid(.degree = 1:2, .nprune = 2:38)
marsTuned <- train(trainingData$x, trainingData$y,
    method = "earth", tuneGrid = marsGrid, trControl = trainControl(method = "cv"))

marsTuned
```

```
## Multivariate Adaptive Regression Spline
##
## 200 samples
##  10 predictors
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
##
## Summary of sample sizes: 180, 180, 180, 180, 180, 180, ...
##
## Resampling results across tuning parameters:
##
##   degree  nprune  RMSE      Rsquared   RMSE SD    Rsquared SD
##   1        2      4.333444  0.3149196  0.6106180  0.16649871
##   1        3      3.820919  0.4404491  0.4967666  0.16657292
##   1        4      3.295404  0.5882392  0.3372786  0.09886721
##   1        5      2.694037  0.7174037  0.3776221  0.12163579
##   1        6      2.344719  0.7782714  0.5489781  0.15029023
##   1        7      1.979736  0.8340402  0.4959329  0.11876164
```

```
##   1        8      1.737149  0.8637278  0.4660162  0.13202708
##   1        9      1.736939  0.8641239  0.5129912  0.13808615
##   1       10      1.725491  0.8675993  0.5461649  0.13584332
##   1       11      1.744738  0.8659244  0.5233297  0.13556243
##   1       12      1.773299  0.8616370  0.5257459  0.13494823
##   1       13      1.784456  0.8601920  0.5228203  0.13432009
##   1       14      1.777995  0.8610093  0.5240934  0.13464491
##   1       15      1.769285  0.8617680  0.5268512  0.13500986
##   1       16      1.774119  0.8612896  0.5258297  0.13475644
##   1       17      1.778343  0.8608839  0.5252999  0.13455450
##   1       18      1.778343  0.8608839  0.5252999  0.13455450
##   1       19      1.778343  0.8608839  0.5252999  0.13455450
##   1       20      1.778343  0.8608839  0.5252999  0.13455450
##   1       21      1.778343  0.8608839  0.5252999  0.13455450
##   1       22      1.778343  0.8608839  0.5252999  0.13455450
##   1       23      1.778343  0.8608839  0.5252999  0.13455450
##   1       24      1.778343  0.8608839  0.5252999  0.13455450
##   1       25      1.778343  0.8608839  0.5252999  0.13455450
##   1       26      1.778343  0.8608839  0.5252999  0.13455450
##   1       27      1.778343  0.8608839  0.5252999  0.13455450
##   1       28      1.778343  0.8608839  0.5252999  0.13455450
##   1       29      1.778343  0.8608839  0.5252999  0.13455450
##   1       30      1.778343  0.8608839  0.5252999  0.13455450
##   1       31      1.778343  0.8608839  0.5252999  0.13455450
##   1       32      1.778343  0.8608839  0.5252999  0.13455450
##   1       33      1.778343  0.8608839  0.5252999  0.13455450
##   1       34      1.778343  0.8608839  0.5252999  0.13455450
##   1       35      1.778343  0.8608839  0.5252999  0.13455450
##   1       36      1.778343  0.8608839  0.5252999  0.13455450
##   1       37      1.778343  0.8608839  0.5252999  0.13455450
##   1       38      1.778343  0.8608839  0.5252999  0.13455450
##   2        2      4.333444  0.3149196  0.6106180  0.16649871
##   2        3      3.820919  0.4404491  0.4967666  0.16657292
##   2        4      3.295404  0.5882392  0.3372786  0.09886721
##   2        5      2.736850  0.7095656  0.3711100  0.11740420
##   2        6      2.450934  0.7622275  0.5169328  0.14290450
##   2        7      2.011364  0.8326857  0.5186990  0.11570458
##   2        8      1.812955  0.8571672  0.5194471  0.13059052
##   2        9      1.694163  0.8714989  0.5700266  0.13866728
##   2       10      1.600453  0.8794668  0.5738009  0.14750489
##   2       11      1.397883  0.9136913  0.3957075  0.07865295
##   2       12      1.346130  0.9180292  0.3767953  0.07900649
##   2       13      1.296523  0.9279181  0.3078141  0.05608037
##   2       14      1.241332  0.9346877  0.2915893  0.05149107
##   2       15      1.236203  0.9362347  0.3077131  0.04997548
##   2       16      1.211930  0.9395960  0.2767813  0.04444609
##   2       17      1.217613  0.9391749  0.2610244  0.04422058
##   2       18      1.208570  0.9399497  0.2656670  0.04446411
##   2       19      1.212076  0.9396013  0.2635088  0.04433802
##   2       20      1.212076  0.9396013  0.2635088  0.04433802
##   2       21      1.212076  0.9396013  0.2635088  0.04433802
##   2       22      1.212076  0.9396013  0.2635088  0.04433802
##   2       23      1.212076  0.9396013  0.2635088  0.04433802
##   2       24      1.212076  0.9396013  0.2635088  0.04433802
```

```
##   2       25      1.212076  0.9396013  0.2635088  0.04433802
##   2       26      1.212076  0.9396013  0.2635088  0.04433802
##   2       27      1.212076  0.9396013  0.2635088  0.04433802
##   2       28      1.212076  0.9396013  0.2635088  0.04433802
##   2       29      1.212076  0.9396013  0.2635088  0.04433802
##   2       30      1.212076  0.9396013  0.2635088  0.04433802
##   2       31      1.212076  0.9396013  0.2635088  0.04433802
##   2       32      1.212076  0.9396013  0.2635088  0.04433802
##   2       33      1.212076  0.9396013  0.2635088  0.04433802
##   2       34      1.212076  0.9396013  0.2635088  0.04433802
##   2       35      1.212076  0.9396013  0.2635088  0.04433802
##   2       36      1.212076  0.9396013  0.2635088  0.04433802
##   2       37      1.212076  0.9396013  0.2635088  0.04433802
##   2       38      1.212076  0.9396013  0.2635088  0.04433802
##
## RMSE was used to select the optimal model using  the smallest value.
## The final values used for the model were nprune = 18 and degree = 2.
```

```
# summary(marsTuned)

# Predict on the test set and see how well it did
marsPredictTuned <- predict(marsTuned, testData$x)
postResample(pred = marsPredictTuned, obs = testData$y)
```

```
##      RMSE  Rsquared
## 1.2667913 0.9356355
```

```
# Inspect the model's use of predictors (notice
# this does not work here as I get an error about
# wrong types; assuming this only works for models
# from using earth()) plotmo(marsTuned,
# caption='Predictors vs Observed in Additive MARS
# Tuned Model')
```

This model does even better, with an R^2 of 0.935!

Support Vector Machine Model

```
# Create a SVM model
svmFit <- ksvm(y ~ ., data = as.data.frame(trainingData),
    kernel = "rbfdot", kpar = "automatic", C = 1, epsilon = 0.1)
```

```
## Using automatic sigma estimation (sigest) for RBF or laplace kernel
```

```
# Show the fit
svmFit
```

```
## Support Vector Machine object of class "ksvm"
##
## SV type: eps-svr  (regression)
##  parameter : epsilon = 0.1  cost C = 1
##
## Gaussian Radial Basis kernel function.
##  Hyperparameter : sigma =  0.075267590304987
##
## Number of Support Vectors : 167
##
## Objective Function Value : -39.9991
## Training error : 0.070171
```

```
summary(svmFit)
```

```
## Length  Class   Mode
##      1   ksvm     S4
```

```
# Predict on the test set and see how well it did
svmPredict <- predict(svmFit, as.data.frame(testData))
postResample(pred = svmPredict, obs = testData$y)
```

```
##      RMSE  Rsquared
## 2.3919302 0.7807456
```

Lets tune a hopefuly better SVM model:

```
# Create a tuned SVM model w/ a radial kernel
svmRTuned.radial <- train(trainingData$x, trainingData$y,
    method = "svmRadial", preProc = c("center", "scale"),
    tuneLength = 14, trControl = trainControl(method = "cv"))

# Show the fit svmRTuned.radial
# summary(svmRTuned.radial)

# Predict on the test set and see how well it did
svmTunedPredict.radial <- predict(svmRTuned.radial,
    testData$x)
print("SVM Model with Radial Kernel")
```

```
postResample(pred = svmTunedPredict.radial, obs = testData$y)



# Create a tuned SVM model w/ a polynomial kernel
svmRTuned.poly <- train(trainingData$x, trainingData$y,
    method = "svmPoly", preProc = c("center", "scale"),
    tuneLength = 3, trControl = trainControl(method = "cv"))

# Show the fit svmRTuned.poly
# summary(svmRTuned.poly)

# Predict on the test set and see how well it did
svmTunedPredict.poly <- predict(svmRTuned.poly, testData$x)
print("SVM Model with Poly Kernel")
postResample(pred = svmTunedPredict.poly, obs = testData$y)
```

```
## [1] "SVM Model with Radial Kernel"
##      RMSE  Rsquared
## 1.9838688 0.8430636
## [1] "SVM Model with Poly Kernel"
##     RMSE Rsquared
## 2.281323 0.801926
```

An SVM model with a radial kernel works best with an R^2 of 0.84. The overall results show that the Nueral Bet model and the MARS model work the best with the models explaining 90%+ of the variance in the data.

Question 7.4

Question

Return to the permeability problem outlined in Exercise 6.2. Train several nonlinear regression models and evaluate the resampling and test set performance.

(a) Which nonlinear regression model gives the optimal resampling and test set performance?

(b) Do any of the nonlinear models outperform the optimal linear model you previously developed in Exercise 6.2? If so, what might this tell you about the underlying relationship between the predictors and the response?

(c) Would you recommend any of the models you have developed to replace the permeability laboratory experiment?

Answer

Like above, we will try out four kinds of models, and we'll see which one does the best and if these models are better than the linear models created in the previous assignment.

Data Pre-Processing

Before creating models, we should make sure the data is properly pre-processed. I will follow the same exact moethodology as per the previous assignment:

```
# Get rid of any predictors that are nero-zero
# variance
nearZero <- nearZeroVar(fingerprints)
fingerprints.filtered <- fingerprints[, -nearZero]

# Filter out highly correlated predictors
correlations <- cor(fingerprints.filtered)
highCorr <- findCorrelation(correlations, cutoff = 0.9)
fingerprints.filtered <- fingerprints.filtered[, -highCorr]

# Split the data into a training and test set
fingerprints.train <- fingerprints.filtered[1:124,
    ]
fingerprints.test <- fingerprints.filtered[1:124, ]
permeability.train <- permeability[1:124, ]
permeability.test <- permeability[1:124, ]
```

The three models will be presented next, with results to follow.

Neural Net Model

```
# Create a Neural Net model
nnetFit <- nnet(fingerprints.train, permeability.train,
    size = 5, decay = 0.01, linout = TRUE, trace = FALSE,
    maxit = 500, MaxNWts = 5 * (ncol(fingerprints.test) +
        1) + 5 + 1)

# Show the fit
nnetFit
```

```
## a 110-5-1 network with 561 weights
## options were - linear output units  decay=0.01
```

```
# summary(nnetFit)

# Predict on the test set and see how well it did
nnetPredict <- predict(nnetFit, fingerprints.test)
postResample(pred = nnetPredict, obs = permeability.test)
```

```
##      RMSE  Rsquared
## 1.7485759 0.9872722
```

KNN Model

```
# Create a KNN model Without trControl =
# trainControl(method = 'cv'), this does even worse
knnFit <- train(x = fingerprints.train, y = permeability.train,
    method = "knn", tuneLength = 10, preProc = c("center",
        "scale"), tuneGrid = data.frame(.k = 1:20),
    trControl = trainControl(method = "cv"))

# Show the model
knnFit
```

```
## k-Nearest Neighbors
##
## 124 samples
## 110 predictors
##
## Pre-processing: centered, scaled
## Resampling: Cross-Validated (10 fold)
##
## Summary of sample sizes: 112, 111, 112, 112, 112, 111, ...
##
## Resampling results across tuning parameters:
##
##   k   RMSE      Rsquared   RMSE SD   Rsquared SD
##    1  12.77031  0.3708302  2.937310  0.2398288
##    2  12.40382  0.3995732  2.875255  0.2650922
##    3  11.84017  0.4117733  2.600842  0.2486208
##    4  11.59904  0.4386195  2.711231  0.2589067
##    5  11.74903  0.4150790  2.838859  0.2554306
##    6  11.46154  0.4414282  3.151197  0.2239655
##    7  11.69611  0.4214861  3.079273  0.2306640
##    8  11.63808  0.4271642  2.949272  0.2269645
##    9  11.64465  0.4262253  2.960472  0.2326115
##   10  11.64188  0.4205205  2.938219  0.2484200
##   11  11.84161  0.4132786  2.990357  0.2617523
##   12  11.93708  0.3954859  2.846443  0.2647243
##   13  12.04157  0.3959101  2.611432  0.2557791
##   14  12.10282  0.3947247  2.580394  0.2612358
##   15  12.01695  0.4057943  2.612941  0.2572831
##   16  12.07023  0.4063249  2.566785  0.2534945
##   17  11.91409  0.4236562  2.661905  0.2716731
##   18  11.99837  0.4119046  2.659684  0.2816954
##   19  12.06877  0.4100561  2.435826  0.2765579
##   20  12.11449  0.4033663  2.446218  0.2685167
##
## RMSE was used to select the optimal model using  the smallest value.
## The final value used for the model was k = 6.
```

```
# Lets make predictions for the test set
knnPred <- predict(knnFit, newdata = fingerprints.test)
postResample(pred = knnPred, obs = permeability.test)
```

```
##      RMSE  Rsquared
## 9.8628852 0.5947749
```

MARS Model

```
# Create a MARS model
marsFit <- earth(fingerprints.train, permeability.train)

# Show the fit
marsFit
```

```
## Selected 16 of 55 terms, and 15 of 110 predictors
## Termination condition: GRSq -10 at 55 terms
## Importance: X157, X503, X698-unused, X6-unused, X141-unused, X371, ...
## Number of terms at each degree of interaction: 1 15 (additive model)
## GCV 113.6165    RSS 7924.748    GRSq 0.5333509    RSq 0.7332244
```

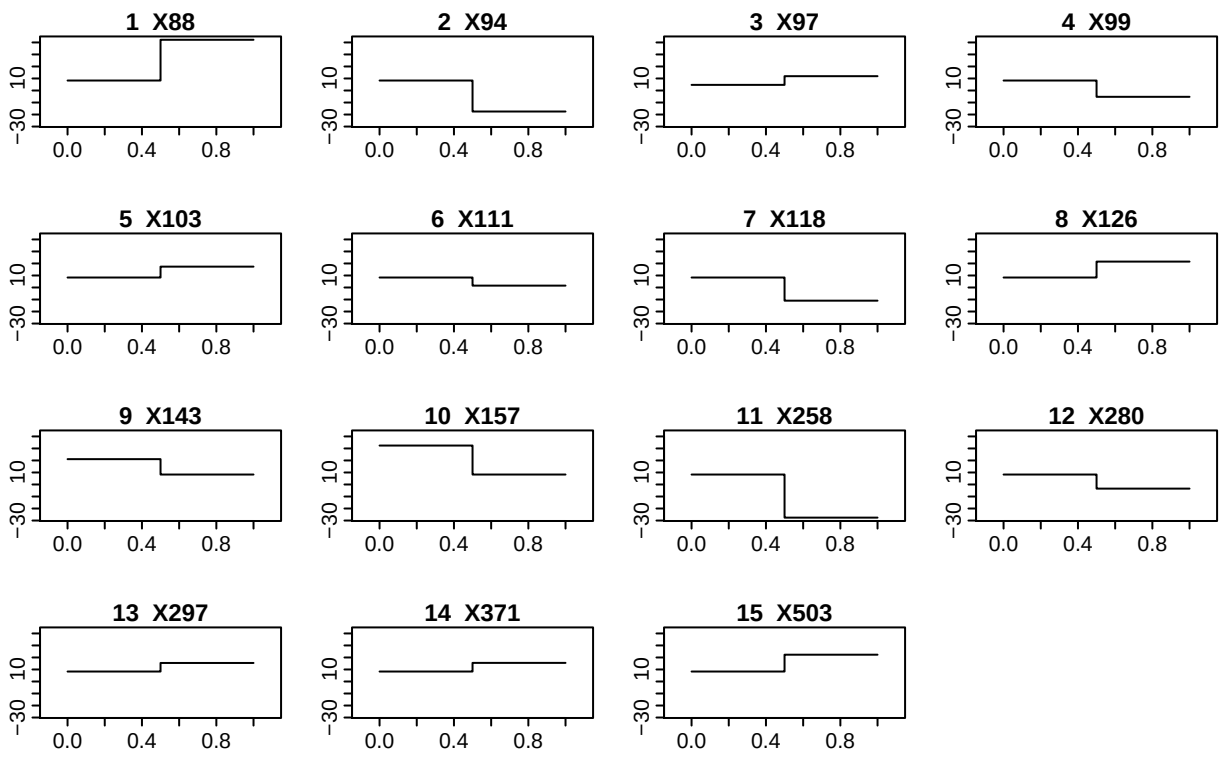
```
# summary(marsFit)

# Predict on the test set and see how well it did
marsPredict <- predict(marsFit, fingerprints.test)
postResample(pred = marsPredict, obs = permeability.test)
```

```
##      RMSE  Rsquared
## 7.9943264 0.7332244
```

```
# Inspect the model's use of predictors
plotmo(marsFit, caption = "Predictors vs Observed in Additive MARS Model")
```

```
##  grid:    X1  X6 X11 X12 X15 X16 X25 X35 X36 X41 X48 X86 X88 X93 X94 X96
##             0 0.5   0   1   0   0   0   0   0   0   0   1   0   0   0   1
##  X97 X98 X99 X103 X111 X118 X121 X125 X126 X141 X143 X146 X157 X158 X159
##  0.5   0   0    0    0    0    0    0    0    0    1    0    1    1    1
##  X182 X221 X225 X226 X229 X230 X231 X235 X237 X238 X242 X251 X257 X258
##     1    1    0    1    1    0    1    1    1    0    0    0    1    0
##  X260 X262 X264 X267 X272 X276 X278 X280 X293 X295 X296 X297 X306 X310
##     1    1    0    1    0    0    0    0    0    0    0    0    0    0
##  X314 X315 X316 X319 X329 X334 X337 X338 X340 X345 X356 X357 X358 X361
##     0    1    0    0    0    0    0    0    0    0    0    0    0    0
##  X362 X366 X371 X372 X374 X377 X499 X503 X507 X508 X509 X510 X511 X512
##     0    0    0    0    0    0    0    0    0    0    0    0    0    0
##  X514 X519 X520 X549 X551 X561 X568 X571 X573 X577 X590 X598 X602 X613
##     0    0    0    0    0    0    0    0    0    0    0    0    0    0
##  X679 X698 X699 X700 X732 X750 X780 X795 X812
##     0    0    0    0    0    0    0    0    0
```


Predictors vs Observed in Additive MARS Model
1 X88
2 X94
3 X97
4 X99
5 X103
6 X111
7 X118
8 X126
9 X143
10 X157
11 X258
12 X280
13 X297
14 X371
15 X503
−30
10
0.0
0.4
0.8

Support Vector Machine Model

```
df.fingerprints.train <- as.data.frame(fingerprints.train)
df.fingerprints.train$y <- permeability.train
df.fingerprints.test <- as.data.frame(fingerprints.test)

svmFit <- ksvm(y ~ ., data = df.fingerprints.train,
    kernel = "rbfdot", kpar = "automatic", C = 1, epsilon = 0.1)
```

```
## Using automatic sigma estimation (sigest) for RBF or laplace kernel
```

```
svmFit
```

```
## Support Vector Machine object of class "ksvm"
##
## SV type: eps-svr  (regression)
##  parameter : epsilon = 0.1  cost C = 1
##
## Gaussian Radial Basis kernel function.
##  Hyperparameter : sigma =  0.00538822397317305
##
## Number of Support Vectors : 98
##
## Objective Function Value : -37.555
## Training error : 0.277195
```

```
# summary(svmFit)

svmPredict <- predict(svmFit, df.fingerprints.test)
postResample(pred = svmPredict, obs = permeability.test)
```

```
##      RMSE  Rsquared
## 8.1820126 0.7476109
```

Lets tune a hopefuly better SVM model:

```
svmRTuned <- train(fingerprints.train, permeability.train,
    method = "svmRadial", preProc = c("center", "scale"),
    tuneLength = 14, trControl = trainControl(method = "cv"))

svmRTuned
```

```
## Support Vector Machines with Radial Basis Function Kernel
##
## 124 samples
## 110 predictors
##
## Pre-processing: centered, scaled
## Resampling: Cross-Validated (10 fold)
##
```

```
## Summary of sample sizes: 112, 110, 112, 112, 112, 112, ...
##
## Resampling results across tuning parameters:
##
##   C        RMSE       Rsquared   RMSE SD   Rsquared SD
##      0.25  12.027841  0.6067874  3.592147  0.2240992
##      0.50  10.930337  0.6194370  3.741335  0.2158559
##      1.00  10.268580  0.6253968  3.722541  0.1988653
##      2.00  10.204321  0.6290063  3.322751  0.1991679
##      4.00   9.924063  0.6314036  2.895832  0.1719067
##      8.00  10.025663  0.6132986  2.505033  0.1429011
##     16.00  10.414601  0.5824339  2.194583  0.1364803
##     32.00  10.560171  0.5735247  2.091467  0.1389528
##     64.00  10.678761  0.5620420  2.085183  0.1505723
##    128.00  10.678881  0.5620398  2.085497  0.1505819
##    256.00  10.678850  0.5620488  2.085614  0.1506134
##    512.00  10.678888  0.5620333  2.085875  0.1506271
##   1024.00  10.678715  0.5620446  2.085703  0.1506189
##   2048.00  10.679173  0.5620031  2.086108  0.1506230
##
## Tuning parameter 'sigma' was held constant at a value of 0.00510706
## RMSE was used to select the optimal model using  the smallest value.
## The final values used for the model were sigma = 0.00510706 and C = 4.
```

```
# summary(svmRTuned)

svmTunedPredict <- predict(svmRTuned, fingerprints.test)
postResample(pred = svmTunedPredict, obs = permeability.test)
```

```
##      RMSE  Rsquared
## 4.5165979 0.9189726
```

Results and Discussions

Lets comapre all the models in one spot:

```
print("Neural Net Performance")
postResample(pred = nnetPredict, obs = permeability.test)
print("KNN Performance")
postResample(pred = knnPred, obs = permeability.test)
print("MARS Performance")
postResample(pred = marsPredict, obs = permeability.test)
print("SVM Performance")
postResample(pred = svmPredict, obs = permeability.test)
print("SVM Tuned Performance")
postResample(pred = svmTunedPredict, obs = permeability.test)
```

```
## [1] "Neural Net Performance"
##      RMSE  Rsquared
## 1.7485759 0.9872722
## [1] "KNN Performance"
##      RMSE  Rsquared
## 9.8628852 0.5947749
## [1] "MARS Performance"
##      RMSE  Rsquared
## 7.9943264 0.7332244
## [1] "SVM Performance"
##      RMSE  Rsquared
## 8.1820126 0.7476109
## [1] "SVM Tuned Performance"
##      RMSE  Rsquared
## 4.5165979 0.9189726
```

It looks like the tuned SVM model the Neural Net model do really well and are the best of the non-linear models checked here.

In my previous assignment, I was able to get a linear model with an R^2 of .91, so I would only recommend the Neural Net model since it seems to have better performance. I might recommend the tuned SVM model as another potential contender because it has a similar R^2 to the best linear model. However, most other linear models had much worse results, and was only able to get near the non-linear models with regularization. This suggests that the data does have some non-linearity to it.