Econ 187 Project 1

2023-05-08

Introduction

In this paper we consider statistical learning methods for classification, then explore some options for regularization in the context of linear regression. We draw heavily from the classic textbook *The Elements of Statistical Learning*, by Hastie et al. We use two different datasets to illustrate these methods in R, one for classification and another for regularization. Each dataset is taken from the UCI Machine Learning Repository, and each is cited as per the citation request provided by the dataset creators.

The dataset we use for the classification methods involves classifying seven different types of dry beans. Images of 13,611 grains of 7 different registered dry beans were taken with a high-resolution camera. Bean images obtained by computer vision system were subjected to segmentation and feature extraction stages, and a total of 16 features (12 dimensions and 4 shape forms) were obtained from the grains. Attributes include area, perimeter, aspect ratio, shape factors, and others.

For the regularization methods, we use a dataset with 81 features extracted from 21263 superconductors along with the critical temperature of each in the 82nd column, the latter of which we attempt to predict. Attributes include mean atomic mass, mean electron affinity, and mean fusion heat, among others.

Classification

Suppse we have a feature set X with p features that we hope to classify into one of K classes in a set of classes Y. Because Y is a set of discrete values, we can divide the feature space into a collection of regions according to the classification within each region. The dividing lines between each region are known as *decision boundaries*, and finding the optimal dividing lines to minimize classification error is our goal.

The only way to truly achieve optimal classification is to determine the class posteriors $\Pr(Y|X)$. If we let π_k denote the prior probability of class $Y = k$, and $f_k(x)$ denote the probability density of X in class k, then Bayes' Theorem tells us that

$$\Pr(Y = k|X = x) = \frac{\pi_k f_k(x)}{\sum_{i=1}^{K} \pi_i f_i(x)}. \tag{1}$$

Note that $\sum_{k=1}^{K} \pi_k = 1$. Clearly, knowing $f_k(x)$ is tantamount to knowing $\Pr(Y = k|X = x)$, but in reality we have to estimate this class density. Suppose we assume that each $f_k(x)$ is Gaussian, or normally distributed, so

$$f_k(x) = \frac{1}{(2\pi)^{p/2}|\mathbf{\Sigma}_k|^{1/2}} e^{-\frac{1}{2}(x-\mu_k)^T \mathbf{\Sigma}_k^{-1}(x-\mu_k)}, \tag{2}$$

where $\mathbf{\Sigma}_k$ is the covariance matrix for class k and μ_k is the mean of class k. In practice, we don't know the parameters of the Gaussian distributions. So, to perform linear discriminant analysis (LDA), we estimate them as follows:

$$
\hat{\pi}_k = \frac{N_k}{N};
$$
$$
\hat{\mu}_k = \sum_{y_i=k} \frac{x_i}{N_k};
$$
$$
\hat{\mathbf{\Sigma}} = \sum_{k=1}^{K} \sum_{y_i=k} \frac{(x_i - \hat{\mu}_k)(x_i - \hat{\mu}_k)^T}{N-K},
$$

where N_k is the number of class-k observations and N is the total number of observations.

Note that, to perform LDA, we have assumed that $\mathbf{\Sigma}_k = \mathbf{\Sigma}$ for all k, so every class has a shared covariance matrix. If we compare two particular classes k and l, the decision boundary between them can be found by looking at the log-ratio, so

$$
\begin{aligned}
\log \frac{\Pr(Y=k|X=x)}{\Pr(Y=l|X=x)} &= \log \frac{f_k(x)}{f_l(x)} + \log \frac{\pi_k}{\pi_l} \\
&= x^T \mathbf{\Sigma}^{-1}(\mu_k - \mu_l) - \frac{1}{2}(\mu_k + \mu_l)^T \mathbf{\Sigma}^{-1}(\mu_k - \mu_l) + \log \frac{\pi_k}{\pi_l},
\end{aligned}
$$

which is a linear equation in x. This implies that the *linear discriminant functions*

$$
\delta_k(x) = x^T \mathbf{\Sigma}^{-1} \mu_k - \frac{1}{2} \mu_k^T \mathbf{\Sigma}^{-1} \mu_k + \log \pi_k \tag{3}
$$

characterize the decision rule, where $\hat{Y}(x) = \text{argmax}_k \delta_k(x)$.

To see how this works in practice, we turn to our dry bean dataset. First, we set up our data.

```r
#load in data
bean <- read.table("C:/Users/Theo/Downloads/Dry_Bean_Dataset.csv", header=TRUE,
   sep=",")

#make sure our target variable is a factor
bean$Class <- as.factor(bean$Class)

#set a seed to ensure our data is reproducible
set.seed(123)

#create 75% training 25% testing split
trainIndex <- createDataPartition(bean$Class, p = .75,
                                  list = FALSE,
                                  times = 1)
training <- bean[trainIndex, ]
testing <- bean[-trainIndex, ]

#set up training control with 10-fold cv
control <- trainControl(method = "cv",
                     number = 10,
                     classProbs = TRUE,
                     verboseIter = FALSE)
```

Now, we can perform LDA, using the `caret` library to set up a 10-fold cross-validation to evaluate our results.

```r
#perform linear discriminant analysis
garbage0 <- capture.output(
lda_fit <- train(Class ~ .,
             data = training,
             method = "lda",
             trControl = control,
             verbose = FALSE))

#predict testing set and create confusion matrix
lda_pred <- predict(lda_fit, newdata = testing)
confusionMatrix(lda_pred, testing$Class)
```

```
## Confusion Matrix and Statistics
##
##             Reference
## Prediction BARBUNYA BOMBAY CALI DERMASON HOROZ SEKER SIRA
##   BARBUNYA      280      0    1        1     1     1    2
##   BOMBAY          0    130    0        0     0     0    0
##   CALI           23      0  390        0    11     0    2
##   DERMASON        0      0    0      752     4     9   37
##   HOROZ           0      0    4        2   446     0    4
##   SEKER           3      0    1       13     0   468    3
##   SIRA           24      0   11      118    20    28  611
##
## Overall Statistics
##
##                Accuracy : 0.905
##                  95% CI : (0.8946, 0.9147)
##     No Information Rate : 0.2606
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.8852
##
##  Mcnemar's Test P-Value : NA
##
## Statistics by Class:
##
##                      Class: BARBUNYA Class: BOMBAY Class: CALI Class: DERMASON
## Sensitivity                  0.84848       1.00000      0.9582          0.8488
## Specificity                  0.99805       1.00000      0.9880          0.9801
## Pos Pred Value               0.97902       1.00000      0.9155          0.9377
## Neg Pred Value               0.98394       1.00000      0.9943          0.9484
## Prevalence                   0.09706       0.03824      0.1197          0.2606
## Detection Rate               0.08235       0.03824      0.1147          0.2212
## Detection Prevalence         0.08412       0.03824      0.1253          0.2359
## Balanced Accuracy            0.92327       1.00000      0.9731          0.9144
##                      Class: HOROZ Class: SEKER Class: SIRA
## Sensitivity                0.9253       0.9249      0.9272
## Specificity                0.9966       0.9931      0.9267
## Pos Pred Value             0.9781       0.9590      0.7525
## Neg Pred Value             0.9878       0.9870      0.9815
## Prevalence                 0.1418       0.1488      0.1938
## Detection Rate             0.1312       0.1376      0.1797
```

```
## Detection Prevalence       0.1341        0.1435       0.2388
## Balanced Accuracy          0.9609        0.9590       0.9269
```

Linear discriminant analysis performs very well, with an accuracy in predicting the test set of 90.5%.

If we no longer assume that each $\mathbf{\Sigma}_k$ is equal, then we have a quadratic term remaining in the discriminant functions. Therefore, we have *quadratic discriminant functions*

$$\delta_k(x) = -\frac{1}{2}\log|\mathbf{\Sigma}_k| - \frac{1}{2}(x-\mu_k)^T\mathbf{\Sigma}_k^{-1}(x-\mu_k) + \log\pi_k, \tag{4}$$

and the decision boundary between each pair of classes k and l is given by $\{x : \delta_k(x) = \delta_l(x)\}$.

We use an almost identical procedure as we did with LDA to execute QDA in R.

```
#perform quadratic discriminant analysis
garbage0 <- capture.output(
qda_fit <- train(Class ~ .,
             data = training,
             method = "qda",
             trControl = control,
             verbose = FALSE))

#predict testing set and create confusion matrix
qda_pred <- predict(qda_fit, newdata = testing)
confusionMatrix(qda_pred, testing$Class)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction BARBUNYA BOMBAY CALI DERMASON HOROZ SEKER SIRA
##   BARBUNYA      296      0    7        1     2     1    7
##   BOMBAY          0    130    0        0     0     0    0
##   CALI           23      0  394        0     7     0    4
##   DERMASON        0      0    0      768     5     7   39
##   HOROZ           0      0    4        2   459     0   14
##   SEKER           3      0    1       17     0   483   11
##   SIRA            8      0    1       98     9    15  584
##
## Overall Statistics
##
##                Accuracy : 0.9159
##                  95% CI : (0.906, 0.925)
##     No Information Rate : 0.2606
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.8985
##
##  Mcnemar's Test P-Value : NA
##
## Statistics by Class:
##
##                      Class: BARBUNYA Class: BOMBAY Class: CALI Class: DERMASON
## Sensitivity                  0.89697       1.00000      0.9681          0.8668
```

```
## Specificity                0.99414       1.00000      0.9886            0.9797
## Pos Pred Value             0.94268       1.00000      0.9206            0.9377
## Neg Pred Value             0.98898       1.00000      0.9956            0.9543
## Prevalence                 0.09706       0.03824      0.1197            0.2606
## Detection Rate             0.08706       0.03824      0.1159            0.2259
## Detection Prevalence       0.09235       0.03824      0.1259            0.2409
## Balanced Accuracy          0.94555       1.00000      0.9783            0.9233
##                      Class: HOROZ Class: SEKER Class: SIRA
## Sensitivity                0.9523       0.9545      0.8862
## Specificity                0.9931       0.9889      0.9522
## Pos Pred Value             0.9582       0.9379      0.8168
## Neg Pred Value             0.9921       0.9920      0.9721
## Prevalence                 0.1418       0.1488      0.1938
## Detection Rate             0.1350       0.1421      0.1718
## Detection Prevalence       0.1409       0.1515      0.2103
## Balanced Accuracy          0.9727       0.9717      0.9192
```

QDA performs slightly better than LDA, with a test-set accuracy of 91.6%.

Next, we fit a *multinomial logistic regression* model to our training data. The motivation behind logistic regression is that we want to model the class posteriors $\Pr(Y|X)$ by linear functions in x, while ensuring that they sum to 1 and remain in $[0,1]$. We can express the model in terms of the log-ratio of the probability for each class, so

$$\begin{aligned}
\log\frac{\Pr(Y=1|X=x)}{\Pr(Y=K|X=x)} &= \beta_{10}+\beta_1^T x \\
\log\frac{\Pr(Y=2|X=x)}{\Pr(Y=K|X=x)} &= \beta_{20}+\beta_2^T x \\
&\dots \\
\log\frac{\Pr(Y=K-1|X=x)}{\Pr(Y=K|X=x)} &= \beta_{(K-1)0}+\beta_{K-1}^T x.
\end{aligned} \tag{5}$$

Exponentiating and rearranging, we find that

$$\begin{aligned}
\Pr(Y=k|X=x) &= \frac{\exp(\beta_{k0}+\beta_k^T x)}{1+\sum_{i=1}^{K-1}\exp(\beta_{i0}+\beta_i^T x)}, \text{ for } k=1,\dots,K-1 \\
\Pr(Y=K|X=x) &= \frac{1}{1+\sum_{i=1}^{K-1}\exp(\beta_{i0}+\beta_i^T x)}.
\end{aligned} \tag{5}$$

```r
#perform multinomial logistic regression
garbage0 <- capture.output(
log_fit <- train(Class ~ .,
             data = training,
             method = "multinom",
             trControl = control,
             verbose = FALSE))

#predict testing set and create confusion matrix
log_pred <- predict(log_fit, newdata = testing)
confusionMatrix(log_pred, testing$Class)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction BARBUNYA BOMBAY CALI DERMASON HOROZ SEKER SIRA
##   BARBUNYA      307      0    7        0     1     1    4
##   BOMBAY          0    130    0        0     0     0    0
##   CALI           12      0  392        0     9     0    2
##   DERMASON        0      0    0      811     5     9   56
##   HOROZ           0      0    4        1   455     0   10
##   SEKER           3      0    1       14     0   487   11
##   SIRA            8      0    3       60    12     9  576
##
## Overall Statistics
##
##                Accuracy : 0.9288
##                  95% CI : (0.9197, 0.9372)
##     No Information Rate : 0.2606
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.9139
##
##  Mcnemar's Test P-Value : NA
##
## Statistics by Class:
##
##                      Class: BARBUNYA Class: BOMBAY Class: CALI Class: DERMASON
## Sensitivity                  0.93030       1.00000      0.9631          0.9153
## Specificity                  0.99577       1.00000      0.9923          0.9722
## Pos Pred Value               0.95938       1.00000      0.9446          0.9205
## Neg Pred Value               0.99253       1.00000      0.9950          0.9702
## Prevalence                   0.09706       0.03824      0.1197          0.2606
## Detection Rate               0.09029       0.03824      0.1153          0.2385
## Detection Prevalence         0.09412       0.03824      0.1221          0.2591
## Balanced Accuracy            0.96303       1.00000      0.9777          0.9438
##                      Class: HOROZ Class: SEKER Class: SIRA
## Sensitivity                0.9440       0.9625      0.8741
## Specificity                0.9949       0.9900      0.9664
## Pos Pred Value             0.9681       0.9438      0.8623
## Neg Pred Value             0.9908       0.9934      0.9696
## Prevalence                 0.1418       0.1488      0.1938
## Detection Rate             0.1338       0.1432      0.1694
## Detection Prevalence       0.1382       0.1518      0.1965
## Balanced Accuracy          0.9694       0.9762      0.9202
```

Multinomial logistic regression performs even better than both LDA and QDA, with a test-set accuracy of 92.9%.

The final classification method we will consider is *k-nearest neighbors*. We classify each point $\mathbf{x}$ in the feature set according to the classifications of the observations "closest" to $\mathbf{x}$ in the feature space. More formally, if we let $N_k(\mathbf{x})$ be the neighborhood of $\mathbf{x}$ defined by the k closest points $\mathbf{x}_i$ according to some metric, then

$$\hat{Y}(x) := \frac{1}{k} \sum_{\mathbf{x}_i \in N_k(\mathbf{x})} y_i. \tag{6}$$

In general, we use the Euclidean L^2 metric, so if $\mathbf{x}^{(1)}$ and $\mathbf{x}^{(2)}$ are observations in the feature space, then

$$d(\mathbf{x}^{(1)}, \mathbf{x}^{(2)}) = \sqrt{\sum_{l=1}^{p} (x_l^{(1)} - x_l^{(2)})^2},$$

where p is the dimension of the feature space.

```
#perform k nearest neighbors
garbage0 <- capture.output(
knn_fit <- train(Class ~ .,
             data = training,
             method = "knn",
             trControl = control))

#predict testing set and create confusion matrix
knn_pred <- predict(knn_fit, newdata = testing)
confusionMatrix(knn_pred, testing$Class)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction BARBUNYA BOMBAY CALI DERMASON HOROZ SEKER SIRA
##   BARBUNYA      164      0  140        0    29     0    3
##   BOMBAY          0    129    0        0     0     0    0
##   CALI          111      1  249        0    36     0    0
##   DERMASON        0      0    0      787    17    87   77
##   HOROZ          43      0   15        1   315     8   53
##   SEKER           0      0    0       47     1   311   46
##   SIRA           12      0    3       51    84   100  480
##
## Overall Statistics
##
##                Accuracy : 0.7162
##                  95% CI : (0.7007, 0.7313)
##     No Information Rate : 0.2606
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.6553
##
##  Mcnemar's Test P-Value : NA
##
## Statistics by Class:
##
##                      Class: BARBUNYA Class: BOMBAY Class: CALI Class: DERMASON
## Sensitivity                  0.49697       0.99231     0.61179          0.8883
## Specificity                  0.94397       1.00000     0.95055          0.9280
## Pos Pred Value               0.48810       1.00000     0.62720          0.8130
## Neg Pred Value               0.94582       0.99969     0.94739          0.9593
## Prevalence                   0.09706       0.03824     0.11971          0.2606
## Detection Rate               0.04824       0.03794     0.07324          0.2315
## Detection Prevalence         0.09882       0.03794     0.11676          0.2847
## Balanced Accuracy            0.72047       0.99615     0.78117          0.9081
```

```
##                         Class: HOROZ Class: SEKER Class: SIRA
## Sensitivity                  0.65353      0.61462     0.7284
## Specificity                  0.95888      0.96752     0.9088
## Pos Pred Value               0.72414      0.76790     0.6575
## Neg Pred Value               0.94368      0.93489     0.9330
## Prevalence                   0.14176      0.14882     0.1938
## Detection Rate               0.09265      0.09147     0.1412
## Detection Prevalence         0.12794      0.11912     0.2147
## Balanced Accuracy            0.80620      0.79107     0.8186
```

We note that kNN performs significantly worse than LDA, QDA, and multinomial logistic regression, with a test-set accuracy of only 71.6%. This suggests that a linear model is more appropriate than a nonlinear model to produce a fit to this data. However, we do note that multinomial logistic regression and QDA both outperform LDA, which suggests that the Bayes decision boundaries between classes are in fact slightly nonlinear; both QDA and multinomial logistic regression have more flexibility than LDA, which allows them to better capture the complexities of the true decision boundaries.

Regularization

We operate in the same setting as above, but now we aim to predict a quantitative variable Y. Consider a linear regression using ordinary least squares (OLS):

$$\hat{\beta}^{\text{OLS}} = \underset{\beta}{\text{argmin}} \sum_{i=1}^{n} (y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j)^2. \tag{7}$$

When we perform OLS regression, issues can arise due to *multicollinearity*, which is when two or more predictors are highly correlated. When this is the case, $X^T X$ is nearly singular, but not quite. Suppose that the two predictors X_a and X_b are highly correlated. OLS will have trouble distinguishing which predictor is responsible for which effects on our response variable Y, as both X_a and X_b point in nearly the same direction (when they are considered as vectors). This leads to unstable coefficient estimates, i.e. coefficient estimates which are highly dependent on the training set we choose. Such high variance will lead to decreased accuracy when predicting the test set.

Shrinkage methods all deal with this problem of multicollinearity by shrinking every coefficient estimate towards zero. This makes the unstable coefficient estimates less problematic, as each coefficient is smaller, decreasing variance. Of course, such a shrinkage increases bias, but we optimize the amount of shrinkage to balance out the bias-variance trade off.

Every shrinkage method shrinks the regression coefficients by imposing a penalty on their size. To perform *ridge regression*, we introduce a hyperparameter λ to penalize the sum-of-squares of the regression coefficients:

$$\hat{\beta}^{\text{ridge}} = \underset{\beta}{\text{argmin}} \left\{ \sum_{i=1}^{n} (y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j)^2 + \lambda \sum_{j=1}^{p} \beta_j^2 \right\} \tag{8}$$

Note that the coefficients are being shrunk towards the origin. Observe that we can rewrite the above as Ordinary Least Squares, but with a size constraint on the parameters, as below:

$$\hat{\beta}^{\text{ridge}} = \underset{\beta}{\text{argmin}} \left\{ \sum_{i=1}^{n} (y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j)^2 \right\},$$
$$\text{subject to } \sum_{j=1}^{p} \beta_j^2 \leq S,$$

to make this size constraint more obvious. To find the ridge regression solutions, we first reparametrize by centering our inputs. Replacing x_{ij} by $x_{ij} - \bar{x}_j$ in (8) yields

$$\hat{\beta}^{\text{ridge}} = \underset{\beta}{\operatorname{argmin}} \Big\{ \sum_{i=1}^{n} (y_i - \beta_0 - \sum_{j=1}^{p} \bar{x}_j \beta_j - \sum_{j=1}^{p} (x_{ij} - \bar{x}_j)\beta_j)^2 + \lambda \sum_{j=1}^{p} \beta_j^2 \Big\}. \tag{9}$$

Now, we can define β^c by

$$\beta_0^c := \beta_0 + \sum_{j=1}^{p} \bar{x}_j \beta_j$$
$$\beta_j^c := \beta_j \text{ for } j = 1, 2, ..., p$$

in order to rewrite (9) as

$$\hat{\beta}^{\text{ridge}} = \underset{\beta^c}{\operatorname{argmin}} \left\{ \sum_{i=1}^{N} [y_i - \beta_0^c - \sum_{j=1}^{p} (x_{ij} - \bar{x}_j)\beta_j^c]^2 + \lambda \sum_{j=1}^{p} (\beta_j^c)^2 \right\}.$$

Now, if we let

$$\tilde{y_i} = y_i - \beta_0^c = y_i - \bar{y},$$
$$\tilde{x}_{ij} = x_{ij} - \bar{x}_j,$$

and if, for convenience, we simply denote β^c as β, then our problem becomes, in matrix form,

$$\min_{\beta} (\tilde{\mathbf{y}} - \tilde{\mathbf{X}}\beta)^T (\tilde{\mathbf{y}} - \tilde{\mathbf{X}}\beta) + \lambda \beta^T \beta.$$

Note that the input matrix $\tilde{\mathbf{X}}$ now has p (rather than $p+1$) columns due to the centering we performed. To solve this, we simply take the derivative with respect to β and set the result equal to zero. We find that

$$\frac{\partial (\tilde{\mathbf{y}} - \beta^T \tilde{\mathbf{X}})^T (\tilde{\mathbf{y}} - \beta^T \tilde{\mathbf{X}})}{\partial \beta} = -2\tilde{\mathbf{X}}^T (\tilde{\mathbf{y}} - \beta^T \tilde{\mathbf{X}}),$$
$$\frac{\partial \lambda \beta^T \beta}{\partial \beta} = 2\lambda\beta,$$

so we derive the first order condition

$$\tilde{\mathbf{X}}^T \tilde{\mathbf{y}} = \tilde{\mathbf{X}}^T \tilde{\mathbf{X}}\beta + \lambda\beta.$$

Solving for β yields the solution

$$\hat{\beta}^{\text{ridge}} = (\tilde{\mathbf{X}}^T \tilde{\mathbf{X}} + \lambda \mathbf{I})^{-1} \tilde{\mathbf{X}}^T \tilde{\mathbf{y}}, \tag{10}$$

where $\mathbf{I}$ is the $p \times p$ identity matrix. To see ridge regression in action, we first set up our data.

```r
#load in data
super <- read.table("C:/Users/Theo/Downloads/superconduct.csv", header=TRUE, sep=",")
super <- na.omit(super)

#set a seed to ensure our data is reproducible
set.seed(123)

#create 75% training 25% testing split
regtrainIndex <- createDataPartition(super$critical_temp, p = .75, list = FALSE, times = 1)
regtrain <- super[regtrainIndex, ]
regtest <- super[-regtrainIndex, ]

#set up new training control
regcontrol <- trainControl(method = "cv", number = 10)
```

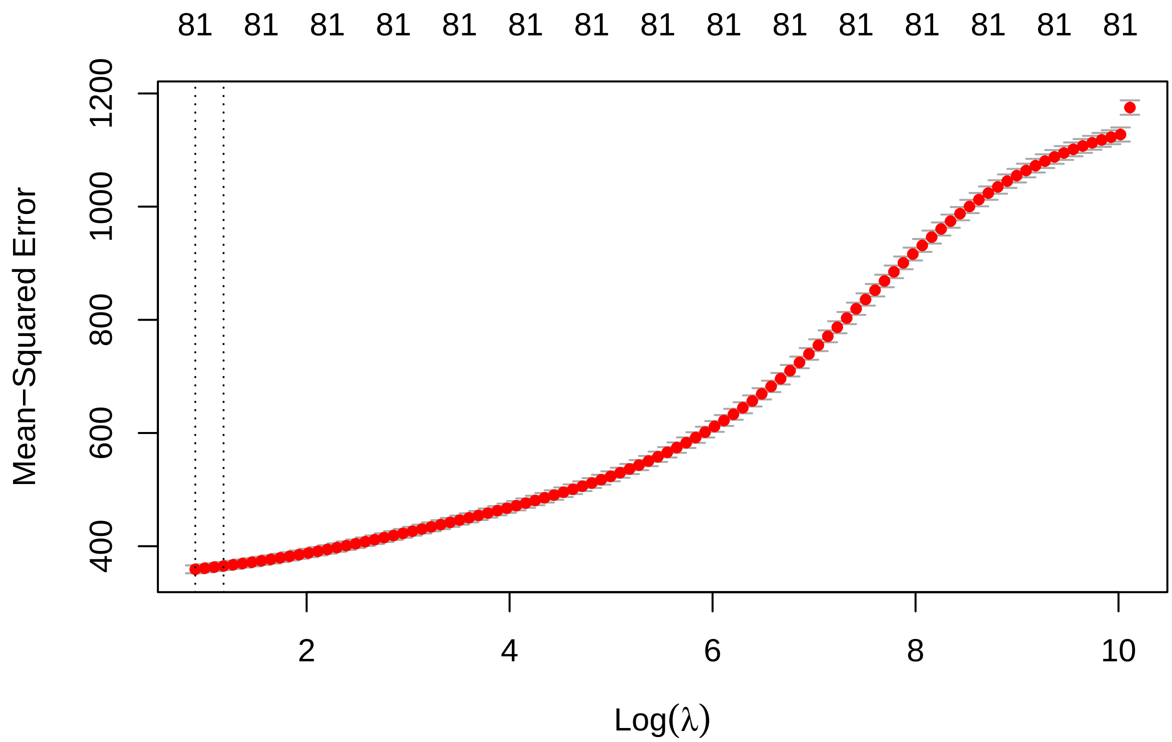
Once again, we can make use of the `caret` library to evaluate our model via a 10-fold cross-validation. We use the `glmnet()` function with $\alpha = 0$, which will be explained below when we encounter the elastic net.

```r
#create grid of possible lambda values
grid = 10^seq(10, -2, length = 100)

#perform ridge regression
ridge_fit <- train(critical_temp ~ .,
                   data = regtrain,
                   method = "glmnet",
                   preProcess = c("center", "scale"), #normalize predictors
                   tuneLength = 25,
                   tuneGrid = expand.grid(alpha = 0, lambda=grid),
                   trControl = regcontrol)

#predict testing data
pred_ridge <- predict(ridge_fit, newdata = regtest)

#use RMSE to evaluate performance
postResample(pred = pred_ridge, obs = regtest$critical_temp)
```

```
##       RMSE   Rsquared        MAE
## 18.9321560  0.6936679 14.5703899
```

We can also perform the exact same task by instead using the `cv.glmnet()` function, which has a cross-validation (with 10 folds by default) built in. Doing so allows us to easily plot the lambda values the function tries against the performance of the model at each value.

```r
#trying it out using cv.glmnet
ridge_new <- cv.glmnet(x=as.matrix(regtrain[,-82]), y=regtrain$critical_temp, alpha=0)

#plot optimal lambda value
plot(ridge_new)
```




```
#evaluate model using RMSE
pred_rnew <- predict(ridge_new, newx=as.matrix(regtest[,-82]), s="lambda.min")
paste("RMSE using cv.glmnet: ", RMSE(pred_rnew, regtest$critical_temp))
```

```
## [1] "RMSE using cv.glmnet:  18.9321559545548"
```

We can see that each implementation of ridge regression yields the same RMSE value when evaluated on the test set of 18.93. We can also check that each method yields a similar optimal lambda value.

```
#optimal lambda with first method
paste("Optimal lambda using caret: ", ridge_fit$bestTune$lambda)
```

```
## [1] "Optimal lambda using caret:  2.00923300256505"
```

```
#optimal lambda with second method
paste("Optimal lambda using cv.glmnet: ", ridge_new$lambda.min)
```

```
## [1] "Optimal lambda using cv.glmnet:  2.4661308119343"
```

Lasso regression is a similar shrinkage method to ridge, but the L^2 ridge penalty is replaced by an L^1 lasso penalty:

$$\hat{\beta}^{\text{lasso}} = \underset{\beta}{\operatorname{argmin}} \left\{ \sum_{i=1}^{n} (y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j)^2 + \lambda \sum_{j=1}^{p} |\beta_j| \right\}. \tag{11}$$

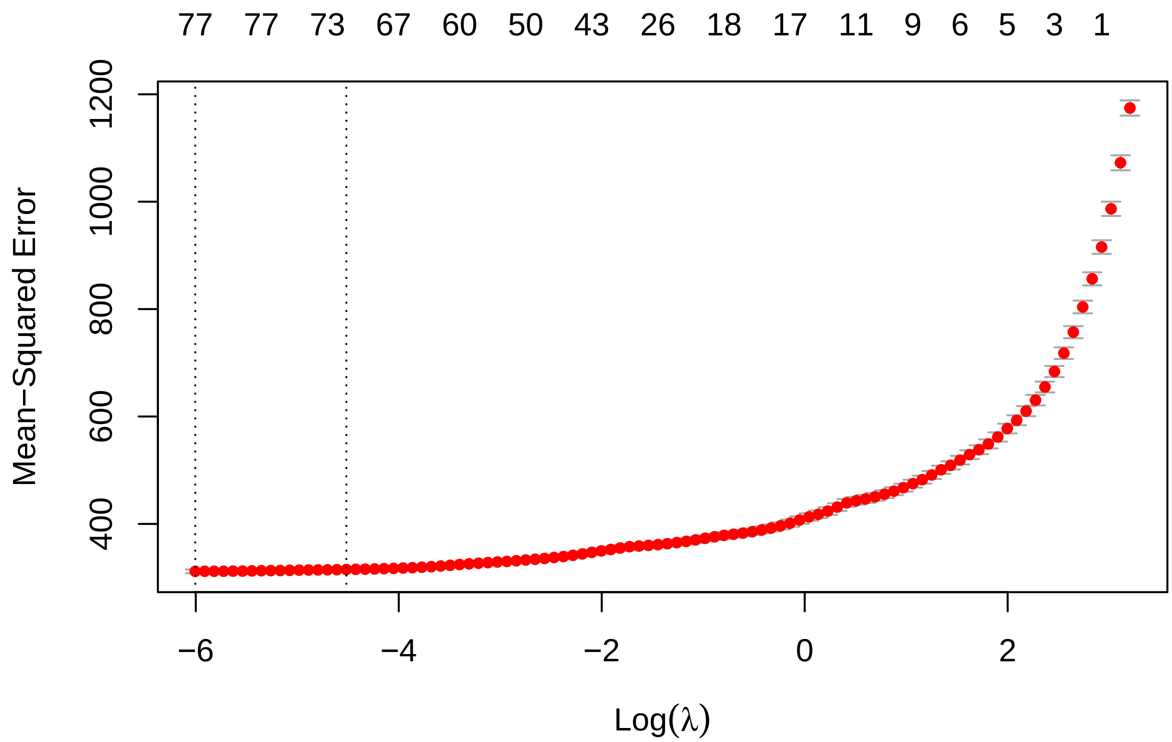
Because this expression is not differentiable everywhere, we cannot find a closed-form solution like we did for ridge. Of course, solutions can be found numerically, but that is beyond the scope of this paper. Again we first use the `caret` library, but this time we set $\alpha = 1$.

```
#perform lasso regression
lasso_fit <- train(critical_temp ~ .,
                   data = regtrain,
                   method = "glmnet",
                   preProcess = c("center", "scale"),
                   tuneLength = 25,
                   tuneGrid = expand.grid(alpha = 1, lambda = grid),
                   trControl = regcontrol)

#predict the testing data
pred_lasso <- predict(lasso_fit, newdata = regtest)

#use RMSE to evaluate performance
postResample(pred = pred_lasso, obs = regtest$critical_temp)
```

```
##      RMSE  Rsquared       MAE
## 17.794077  0.728729 13.438282
```

We find that lasso regression using `caret` yields an RMSE of 17.79 when evaluated on the test set. Again, we also use `cv.glmnet`:

```
#use cv.glmnet to fit a lasso model
lasso_new <- cv.glmnet(x=as.matrix(regtrain[,-82]), y=regtrain$critical_temp, alpha=1)

#plot the lambdas vs performance
plot(lasso_new)
```




```
#evaluate performance on the test set
pred_lnew <- predict(lasso_new, newx=as.matrix(regtest[,-82]), s="lambda.min")
paste("RMSE for lasso: ", RMSE(pred_lnew, regtest$critical_temp))
```

```
## [1] "RMSE for lasso:  17.724091333874"
```

Again, `cv.glmnet` yields an extremely similar RMSE value of 17.72. Across both implementations, lasso outperforms ridge in terms of RMSE.

We now consider an *elastic net regression*, which combines the penalties of ridge and lasso by introducing a new hyperparameter α which determines how much impact each penalty term has.

$$\hat{\beta}^{\text{enet}} = \underset{\beta}{\operatorname{argmin}}\left\{\sum_{i=1}^{n}(y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j)^2 + \lambda\Big(\frac{1-\alpha}{2}\sum_{j=1}^{p}\beta_j^2 + \alpha\sum_{j=1}^{p}|\beta_j|\Big)\right\}. \tag{12}$$

We note that when $\alpha = 0$, elastic net reduces to ridge regression, and similarly, when $\alpha = 1$, we simply have a lasso regression. By now, it is probably clear that the `glmnet()` function we were using above actually performs an elastic net regression, but we were modifying it so it would yield ridge and lasso regressions instead.

The implementation of elastic net is thus very similar to what we've already done, but since elastic net requires optimizing two hyperparameters (both λ and α) simultaneously, we use the `caret` library.

```
#perform elastic net regression
enet_fit <- train(critical_temp ~ .,
```

```r
                     data = regtrain,
                     method = "glmnet",
                     preProcess = c("center", "scale"),
                     tuneLength = 25,
                     trControl = regcontrol)

#predict testing data
pred_enet <- predict(enet_fit, newdata = regtest)

#use RMSE to evaluate performance
postResample(pred = pred_enet, obs = regtest$critical_temp)
```

```
##       RMSE   Rsquared        MAE
## 17.7613062  0.7297279 13.4112253
```

We find that elastic net yields an RMSE of 17.76 when evaluated on the test set, so it performs essentially the same as lasso and better than ridge.

Finally, we introduce *principal component regression (PCR)*, a method of regularization that doesn't involve shrinkage. Principal component regression involves first performing principal component analysis (PCA) to address multicollinearity, then regressing on the chosen principal components. We will explain how to implement PCA, and it will become clear how this eliminates multicollinearity by contruction.

To perform PCA, we first assume that the matrix of predictors X is mean-centered and standardized. Then we find the correlation matrix

$$\mathbf{Q} = \frac{1}{N-1}\mathbf{X}^T\mathbf{X}$$

Next, we diagonalize this matrix, assuming $\mathbf{X}$ has full column-rank:

$$\mathbf{Q} = \mathbf{W}\mathbf{\Lambda}\mathbf{W}^T,$$

where $\mathbf{\Lambda}$ is the diagonal matrix of eigenvalues of $\mathbf{Q}$. Now, we order the eigenvectors based on the size of their corresponding eigenvalues, and we choose the M eigenvectors with the largest eigenvalues.

Finally, we project the data onto the M eigenvectors that we chose:

$$\mathbf{T}_M = \mathbf{X}\mathbf{W}_M \tag{13}$$

Note that the transformation $\mathbf{T}_M$ maps a data vector from the feature space to a new space of M variables that are uncorrelated. The column vectors of $\mathbf{T}_M$ are called *principal components*, and we use these as the new predictors on which we will regress.

Notably, we can think of ridge regression as being a kind of smooth version of PCA. To see this, we introduce the *singular value decomposition (SVD)* of the mean-centered predictor matrix

$$\mathbf{X} = \mathbf{U}\mathbf{S}\mathbf{V}^T, \tag{14}$$

where $\mathbf{S}$ is a diagonal matrix with diagonal elements s_i. We plug this into our ridge solution (10) to find that

$$\hat{\mathbf{y}}^{\text{ridge}} = \mathbf{X}\hat{\beta}^{\text{ridge}} = \mathbf{X}(\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^T\mathbf{y}$$
$$= \mathbf{U}\text{diag}\left\{\frac{s_i^2}{s_i^2 + \lambda}\right\}\mathbf{U}^T\mathbf{y}.$$

Similarly, if we substitute the SVD (14) into our expression for PCA (13), we find that

$$\begin{aligned}\mathbf{T} &= \mathbf{X}\mathbf{W} \\ &= \mathbf{U}\mathbf{S}\mathbf{W}^T\mathbf{W} \\ &= \mathbf{U}\mathbf{S},\end{aligned}$$

so we have that $\mathbf{T}_M = \mathbf{U}_M\mathbf{S}_M$. Thus, we can write

$$\hat{\mathbf{y}}^{\text{PCR}} = \mathbf{X}^{\text{PCA}}\hat{\beta}^{\text{PCA}} = \mathbf{U}\text{diag}\{1_1, 1_2, \ldots, 1_M, 0, \ldots, 0\}\mathbf{U}^T\mathbf{y}. \tag{15}$$

We now implement PCR in R. First, we perform PCA using the function `prcomp()`. We make sure to standardize our predictors by including `scale. = TRUE` so that each predictor has zero mean and unit variance. This was done automatically by the `glmnet()` function, but we must standardize the predictors when using shrinkage methods as well.

```r
#perform pca
pcs <- prcomp(super[,-82], scale. = TRUE)
summary(pcs)
```

```
## Importance of components:
##                           PC1    PC2     PC3     PC4     PC5     PC6     PC7
## Standard deviation     5.6156 2.9139 2.77708 2.53086 2.18279 1.75174 1.71290
## Proportion of Variance 0.3893 0.1048 0.09521 0.07908 0.05882 0.03788 0.03622
## Cumulative Proportion  0.3893 0.4941 0.58935 0.66843 0.72725 0.76513 0.80136
##                            PC8     PC9    PC10    PC11    PC12    PC13    PC14
## Standard deviation     1.58643 1.38293 1.26573 1.21695 1.08695 0.97701 0.89935
## Proportion of Variance 0.03107 0.02361 0.01978 0.01828 0.01459 0.01178 0.00999
## Cumulative Proportion  0.83243 0.85604 0.87582 0.89410 0.90869 0.92047 0.93046
##                           PC15    PC16    PC17    PC18    PC19    PC20    PC21
## Standard deviation     0.89208 0.79563 0.76303 0.66348 0.62570 0.55602 0.49481
## Proportion of Variance 0.00982 0.00782 0.00719 0.00543 0.00483 0.00382 0.00302
## Cumulative Proportion  0.94028 0.94810 0.95529 0.96072 0.96555 0.96937 0.97239
##                           PC22    PC23    PC24    PC25    PC26   PC27    PC28
## Standard deviation     0.48177 0.45580 0.40959 0.39968 0.38845 0.3711 0.33985
## Proportion of Variance 0.00287 0.00256 0.00207 0.00197 0.00186 0.0017 0.00143
## Cumulative Proportion  0.97526 0.97782 0.97989 0.98187 0.98373 0.9854 0.98686
##                           PC29    PC30    PC31    PC32    PC33    PC34    PC35
## Standard deviation     0.31984 0.30537 0.28801 0.27892 0.27287 0.24129 0.23554
## Proportion of Variance 0.00126 0.00115 0.00102 0.00096 0.00092 0.00072 0.00068
## Cumulative Proportion  0.98812 0.98927 0.99029 0.99125 0.99217 0.99289 0.99358
##                           PC36    PC37    PC38    PC39    PC40    PC41    PC42
## Standard deviation     0.22426 0.21502 0.19983 0.18811 0.18503 0.16253 0.15744
## Proportion of Variance 0.00062 0.00057 0.00049 0.00044 0.00042 0.00033 0.00031
## Cumulative Proportion  0.99420 0.99477 0.99526 0.99570 0.99612 0.99645 0.99675
```

```
##                           PC43    PC44    PC45    PC46   PC47    PC48    PC49
## Standard deviation     0.14428 0.13868 0.13456 0.13208 0.1262 0.12326 0.12100
## Proportion of Variance 0.00026 0.00024 0.00022 0.00022 0.0002 0.00019 0.00018
## Cumulative Proportion  0.99701 0.99725 0.99747 0.99769 0.9979 0.99807 0.99825
##                           PC50    PC51    PC52    PC53    PC54    PC55    PC56
## Standard deviation     0.11914 0.11255 0.11162 0.10131 0.09863 0.09771 0.09242
## Proportion of Variance 0.00018 0.00016 0.00015 0.00013 0.00012 0.00012 0.00011
## Cumulative Proportion  0.99843 0.99858 0.99874 0.99886 0.99898 0.99910 0.99921
##                           PC57    PC58    PC59    PC60    PC61    PC62    PC63
## Standard deviation     0.08503 0.08118 0.08045 0.07627 0.07243 0.06789 0.05995
## Proportion of Variance 0.00009 0.00008 0.00008 0.00007 0.00006 0.00006 0.00004
## Cumulative Proportion  0.99930 0.99938 0.99946 0.99953 0.99959 0.99965 0.99970
##                           PC64    PC65    PC66    PC67    PC68    PC69    PC70
## Standard deviation     0.05968 0.05650 0.05349 0.05107 0.04773 0.04298 0.04084
## Proportion of Variance 0.00004 0.00004 0.00004 0.00003 0.00003 0.00002 0.00002
## Cumulative Proportion  0.99974 0.99978 0.99981 0.99985 0.99988 0.99990 0.99992
##                           PC71    PC72    PC73    PC74    PC75    PC76    PC77
## Standard deviation     0.03832 0.03676 0.03457 0.02734 0.02483 0.02113 0.01819
## Proportion of Variance 0.00002 0.00002 0.00001 0.00001 0.00001 0.00001 0.00000
## Cumulative Proportion  0.99994 0.99995 0.99997 0.99998 0.99999 0.99999 0.99999
##                           PC78    PC79     PC80     PC81
## Standard deviation     0.01366 0.01096 0.008603 0.007042
## Proportion of Variance 0.00000 0.00000 0.000000 0.000000
## Cumulative Proportion  1.00000 1.00000 1.000000 1.000000
```

We see that using the first 17 components captures 95% of the variance in our feature set, while using the first 30 components captures 99% of the variance.

```
#define new training control using first 17 components of pca
newcont <- trainControl(method = "cv", number = 10,
                     preProcOptions = list(thresh = 0.95, pcaComp = 17))

#train regression model with principal components as predictors
pcr <- train(critical_temp ~ ., data = regtrain, method = "lm", preProcess = c("center", "scale", "pca")

#predict on testing data
pred_pcr <- predict(pcr, newdata = regtest)

#evaluate performance using RMSE
postResample(pred = pred_pcr, obs = regtest$critical_temp)
```

```
##       RMSE   Rsquared        MAE
## 21.6264414  0.5992832 17.0645484
```

We find that the PCR fit has an RMSE of 21.63 when evaluated on the test set.

After considering all four regularization models, we conclude that lasso performed the best on our data set; elastic net performed essentially the same, which makes sense because elastic net reduces to lasso when we set $\alpha = 1$. PCR performed the worst, which also makes sense because the focus of PCA is to eliminate multicollinearity, rather than optimizing for accuracy in predicting the test set.

Citations

Hastie, T., Tibshirani, R., Friedman, J. (2009). The Elements of Statistical Learning. Springer Series in Statistics. Springer, New York, NY. [https://doi.org/10.1007/978-0-387-84858-7_14]

Hamidieh, Kam, A data-driven statistical model for predicting the critical temperature of a superconductor, Computational Materials Science, Volume 154, November 2018, Pages 346-354, [https://doi.org/10.1016/j.commatsci.2018.07.052]

KOKLU, M. and OZKAN, I.A., (2020), Multiclass Classification of Dry Beans Using Computer Vision and Machine Learning Techniques. Computers and Electronics in Agriculture, 174, 105507. DOI: [https://doi.org/10.1016/j.compag.2020.105507]