Implementation of Linear Discriminant Analysis

Michael Frasco

April 28, 2015

Results: When using linear discriminant analysis, I find that the best model achieves an error of less than 15% on the unseen test data. My model, trained on 4,000 images, included a smoothing parameter to make the covariance matrix of model invertible. I chose this smoothing parameter by cross validating my model on a set of 1,000 images. I found that the best smoothing parameter was a value around 0.2.

Below is the code that I wrote to implement this model. At the bottom of the code, you can see a plot of the different cross-validated errors that were used to select the optimal smoothing parameter.

```
PrepareData <- function(training.data, training.label, p) {
      # Input: training data, training labels, and the proportion of
            # data that should be used to train
      # Output: a training set and a development data set

      training.data <- cbind(training.data, training.label)
      train.index <- createDataPartition(training.label, p=p,list=FALSE)
      train.data <- training.data[train.index,]
      dev.data <- training.data[-train.index,]
      return(list(train.data, dev.data))
}

GenerateLists <- function(train.data) {
      # Input: training data
      # output: a list of lists. The first list contains the column means for
            # the subset of the training data that belong to each digit class.
            # The second list contains the covariance matrices.
            # And the third list contains the proportions of each digit class
            # subset within the training data.

      mean.list <- list()
      cov.list <- list()
      pro.list <- list()
      for(label in 0:9) {
            data.subset <- subset(train.data, train.data[,401]==label)
            data.mean <- as.vector(colMeans(data.subset)[1:400])
            data.cov <- cov(data.subset[,1:400])
            mean.list[[label+1]] <- data.mean
            cov.list[[label+1]] <- data.cov
            pro.list[[label+1]] <- nrow(data.subset) / nrow(train.data)
      }
      return(list(mean.list, cov.list, pro.list))
}

PoolCov <- function(cov.list, pro.list) {
      # Input: a list of covariance matrices for each digit class and the
            # proportions of each class in the training data
      # Output: A pooled covariance matrix.

      d <- nrow(cov.list[[1]]) # Get dimension for covariance matrix
```

```
      pooled.cov <- matrix(0, nrow=d, ncol=d)
      for(i in 1:length(cov.list)) {
            cov.i <- cov.list[[i]]
            cov.i <- cov.i * pro.list[[i]]
            pooled.cov <- pooled.cov + cov.i
      }
      return(pooled.cov)
}

SmoothCov <- function(pooled.cov, lambda, dVal) {
      # Input: a pooled covariance matrix, the smoothing parameter lambda,
            # and the value that should be used on the diagonal of the
            # smoothing matrix
      # Output: a smoothed (i.e. inverible) covariance matrix

      if(abs(det(pooled.cov)) < 0.001) { # near singular
            smoothing.matrix <- lambda * diag(dVal, nrow = nrow(pooled.cov))
            smoothed.cov <- (1 - lambda) * pooled.cov + smoothing.matrix
            return(smoothed.cov)
      } else {
            return(pooled.cov)
      }
}

GetCoefs <- function(mean.list, inverted.cov, pro.list) {
      # input: column means for each digit class, the inverse of the smoothed
            # covariance matrix, and the proportions for each digit class
      # output: based on the derivations of LDA, we create two lists of
            # coefficients that will be used to evaluate each data point

      a.list <- list()
      b.list <- list()
      num.digits <- length(mean.list)
      for(i in 1:num.digits) {
            # We perform the same calculations for each digit class
            mean.i <- mean.list[[i]]
            a <- t(mean.i) %*% inverted.cov
            a.list[[i]] <- as.numeric(a)

            b <- - (1/2) * t(mean.i) %*% inverted.cov %*% mean.i +
                  log(pro.list[[i]])
            b.list[[i]] <- as.numeric(b)
      }
      return(list(a.list, b.list))
}

BuildModel <- function(train.data, lambda, dVal) {
      # input: the train data, the value of the smoothing parameter
            # lambda, and the value on the diagonal of the smoothing matrix
      # output: two lists of coefficients for each class

      info.lists <- GenerateLists(train.data)
      mean.list <- info.lists[[1]]
```

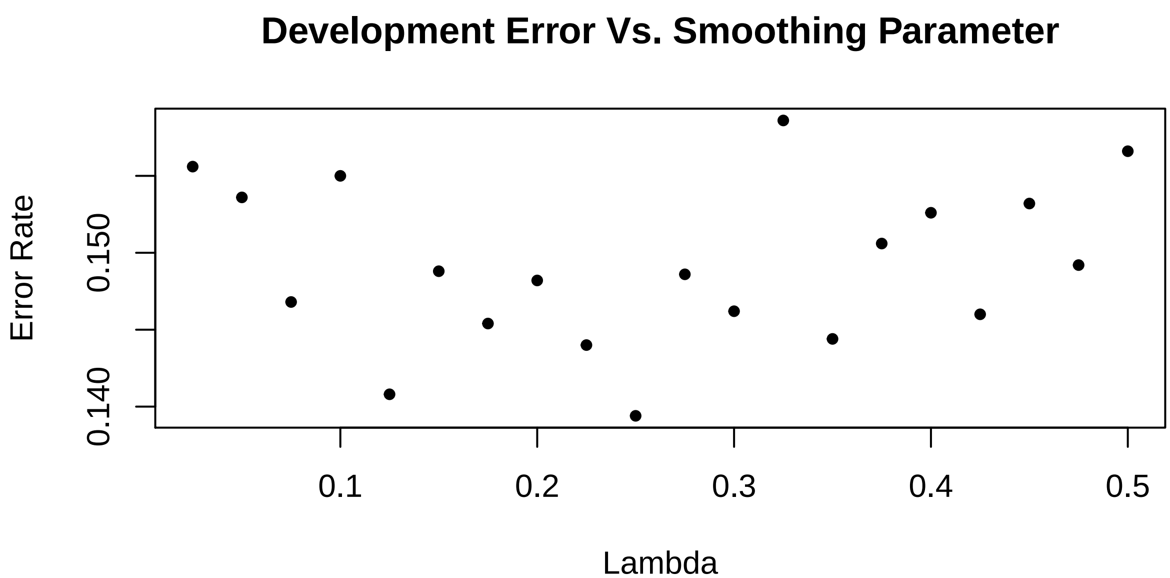
```
      cov.list <- info.lists[[2]]
      pro.list <- info.lists[[3]]

      pooled.cov <- PoolCov(cov.list, pro.list)
      smoothed.cov <- SmoothCov(pooled.cov, lambda, dVal)
      inverted.cov <- solve(smoothed.cov) # gets the matrix inverse

      abLists <- GetCoefs(mean.list, inverted.cov, pro.list)
      return(abLists)
}

GetErrorRate <- function(data, a.list, b.list) {
      # input: a data set and the two lists of coefficients from the model
      # output: an error rate on the data set set

      numErrors <- 0
      for(i in 1:nrow(data)){
            x <- as.vector(data[i,1:400])
            y <- as.numeric(data[i,401])
            max.value <- -Inf
            max.label <- -1
            num.digits <- length(a.list)
            for(k in 1:num.digits) {
                  # We want to classify each point in the development set
                        # to whichever class maximizes the expression below
                  k.value <- a.list[[k]] %*% x + b.list[[k]]
                  if(k.value > max.value) {
                        max.value <- k.value
                        max.label <- k - 1
                        # since the first value corresponds to the digit "0"
                  }
            }
            if(max.label != y) {
                  # Did we make the wrong prediction
                  numErrors = numErrors + 1
            }
      }
      return(numErrors / nrow(data))
}
```

```
p <- 0.8
lValues <- seq(0.025, 0.5, by=0.025)
dVal <- 0.25
error.mean <- c()
for (lambda in lValues) {
      # loop over different lambda values
      avg.error <- 0
      for(trial in 1:5) {
            # repeat process five times for each parameter
            data.sets <- PrepareData(training.data, training.label, p)
            train.data <- data.sets[[1]]
            dev.data <- data.sets[[2]]
```

```
            abLists <- BuildModel(train.data, lambda, dVal)
            a.list <- abLists[[1]]
            b.list <- abLists[[2]]
            error <- GetErrorRate(dev.data, a.list, b.list)
            avg.error <- avg.error + error
      }
      error.mean <- append(error.mean, avg.error / 5)
}
print(error.mean)
```

```
##  [1] 0.1556 0.1536 0.1468 0.1550 0.1408 0.1488 0.1454 0.1482 0.1440 0.1394
## [11] 0.1486 0.1462 0.1586 0.1444 0.1506 0.1526 0.1460 0.1532 0.1492 0.1566
```

The numbers printed above are the error rates that correspond to each of the values for lambda that were used in the code directly above. As you can see, a value of lambda near 0.2 minimizes the error rate.




```
test.data.label <- cbind(test.data, test.label)
p <- 0.8
lambda <- lValues[which.min(error.mean)]
dVal <- 0.25
data.sets <- PrepareData(training.data, training.label, p)
train.data <- data.sets[[1]]
dev.data <- data.sets[[2]]
abLists <- BuildModel(train.data, lambda, dVal)
testError <- GetErrorRate(test.data.label, abLists[[1]], abLists[[2]])
print(paste(lambda))
```

```
## [1] "0.25"
```

```
print(paste(testError))
```

```
## [1] "0.1477"
```