EM Algorithm

Michael Frasco

May 7, 2015

Results: On the unseen test data, we achieve an accuracy around 90% on the test data. We selected the number of mixture components by choosing the number which gave the lowest cross validated accuracy subject to the condition that each mixture has at least one picture assigned to it. This resulted in using 8 mixture components for each digit class.

Below you will find plots of our log likelihood function and images that were created as a result of the EM algorithm.

Prepare data

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

      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))
}
```

LogLikelihood

```
ComputeLogLik <- function(data, weights, prob.mat, pi.vec, num.mix) {
      # input: data, a matrix of pixel probabilities for each class, the
            # marginal probabilities for each class, and the number of mixtures
      # output: the log likelihood as calculated with the derived formula.

      logLik <- 0
      for(i in 1:nrow(data)) {
            x <- data[i, ]
            weight.vec <- weights[i, ] # to get the expected value
            row.sum <- 0
            for(m in 1:num.mix) {
                  prob <- dbinom(x, 1, prob.mat[,m]) # here we use the built
                        # in function dbinom which returns a vector of
                        # binomial probabilities
                  new.term <- weight.vec[m] * (sum(log(prob)) + log(pi.vec[m]))
                  row.sum <- row.sum + new.term
            }
            logLik <- logLik + row.sum
      }
      return(logLik)
}
```

WEIGHTS

```
ComputeWeightRow <- function(x, prob.mat, pi.vec, num.mix) {
      # input: a single picture, the pixel probability matrix, the marginal
            # class probabilities, and the number of mixtures
      # output: a row to be added to the weight matrix

      row.i <- rep(0, num.mix)
      x <- as.vector(x)
      for(m in 1:num.mix) {
            # take advantage of the fact that we can use the same x for
                  # all values of m
            prob <- dbinom(x, 1, prob.mat[,m]) # returns a vector of binomial
                  # probabilities for each pixel in x
            row.i[m] <- sum(log(prob)) + log(pi.vec[m])
      }
      max.val <- max(row.i)
      row.i <- exp(row.i - max.val)
      row.i.sum <- sum(row.i)
      return(row.i / row.i.sum) # use the given formula to return a whole row
}

ComputeWeightMatrix <- function(data, prob.mat, pi.vec, num.mix) {
      # input: a single picture, the pixel probability matrix, the marginal
            # class probabilities, and the number of mixtures
      # output: the weight matrix

      weights <- matrix(0, nrow = nrow(data), ncol = num.mix)
      for(i in 1:nrow(data)) {
            # build the weight matrix one row at a time
            weights[i, ] <- ComputeWeightRow(data[i, ], prob.mat,
                                             pi.vec, num.mix)
      }
      return(weights)
}
```

PROBS

```
UpdateProbCol <- function(weight.vec, data, phi, sigma) {
      # input: a column from the weight matrix, the data, and parameters
      # output: a column in the pixel probability matrix

      col <- as.vector(t(data) %*% weight.vec) # matrix product
      weight.sum <- sum(weight.vec)
      return((col + phi - 1) / (weight.sum + phi + sigma - 2))
}

UpdateProbMat <- function(weights, data, phi, sigma, num.mix) {
      # input: the weights matrix, the data, parameters, and number of mixtures
      # output: the pixel probability matrix

      prob.mat <- matrix(0, nrow=ncol(data), ncol=num.mix)
      for(m in 1:num.mix) {
            # we take advantage of the fact that the same column in the
                  # weight matrix can be used to get an entire column
```

```
            prob.mat[, m] <- UpdateProbCol(weights[,m], data, phi, sigma)
      }
      return(prob.mat)
}
```

PI

```
UpdatePi <- function(weight.vec, N, alpha, num.mix) {
      # input: a weight vector, the number of training examples
      # output: a single element in the marginal probability vector

      weight.sum <- sum(weight.vec)
      return((weight.sum + alpha - 1) / (N + num.mix * (alpha - 1)))
}

UpdatePiVec <- function(weights, alpha, num.mix) {
      # input: the weights, parameter, and the number of mixtures
      # output: the marginal probability vector for digit classes

      pi.vec <- rep(0, num.mix)
      N <- nrow(weights)
      for(m in 1:num.mix) {
            pi.vec[m] <- UpdatePi(weights[,m], N, alpha, num.mix)
      }
      return(pi.vec)
}
```

INITIALIZE: All of these functions are identical to the ones above. They just use the initial assignment, instead of using the weights matrix.

```
InitializeProbCol <- function(indicator, data, phi, sigma) {
      col <- as.vector(t(data) %*% indicator)
      indicator.sum <- sum(indicator)
      return((col + phi - 1) / (indicator.sum + phi + sigma - 2))
}

InitializeProbMat <- function(init.assn, data, phi, sigma, num.mix) {
      prob.mat <- matrix(0, nrow=ncol(data), ncol=num.mix)
      for(m in 1:num.mix) {
            indicator <- init.assn == m
            prob.mat[, m] <- InitializeProbCol(indicator, data, phi, sigma)
      }
      return(prob.mat)
}

InitializePi <- function(indicator, N, alpha, num.mix) {
      indicator.sum <- sum(indicator)
      return((indicator.sum + alpha - 1) / (N + num.mix * (alpha - 1)))
}

InitializePiVec <- function(init.assn, alpha, num.mix) {
      pi.vec <- rep(0, num.mix)
      N <- length(init.assn)
```

```
      for(m in 1:num.mix) {
            indicator <- init.assn == m
            pi.vec[m] <- InitializePi(indicator, N, alpha, num.mix)
      }
      return(pi.vec)
}

InitializeWeights <- function(N, M) {
      # also initialize random weights so that each row sums to one
      weights <- matrix(0, nrow=N, ncol=M)
      for(i in 1:N) {
            weight.row <- runif(M)
            weight.row <- weight.row / sum(weight.row)
            weights[i,] <- weight.row
      }
      return(weights)
}
```

RUN EVERYTHING

```
runEM <- function(data, num.mix, phi, sigma, alpha) {
      # input: a data set, the number of mixtures to use, and parameters
      # output: log likelihood values, weights, pixel probability matrix,
            # the marginal class probabilities

      # initialize using random assignments
      init.assn <- sample(num.mix, nrow(data), replace=TRUE)
      prob.mat <- InitializeProbMat(init.assn, data, phi, sigma, num.mix)
      pi.vec <- InitializePiVec(init.assn, alpha, num.mix)
      weights <- InitializeWeights(nrow(data), num.mix)

      log.lik.values <- c()
      index <- 0
      keep.going <- TRUE
      while(keep.going & index < 10) { # perform no more than 10 iterations
            index <- index + 1
            log.lik.values[index] <- ComputeLogLik(data, weights,
                                                   prob.mat, pi.vec, num.mix)

            weights <- ComputeWeightMatrix(data, prob.mat, pi.vec, num.mix)
            prob.mat <- UpdateProbMat(weights, data, phi, sigma, num.mix)
            pi.vec <- UpdatePiVec(weights, alpha, num.mix)

            # stopping conditions for EM algorithm
            if(index > 2) {
                  curr.val <- log.lik.values[index]
                  old.val <- log.lik.values[index - 1]
                  if(abs((curr.val - old.val) / old.val) < 0.001) {
                        # stop if likelihood increased by less than 0.1%
                        keep.going < TRUE
                  }
            }
      }
```

```
      return(list(log.lik.values, weights, prob.mat, pi.vec))
}

CreateTenMixtures <- function(train.data, num.mix, phi, sigma, alpha) {
      # input: training data, number of mixtures, and parameters
      # output: run the algorithm on each subset of the data

      ten.EMs <- list()
      for(label in 0:9) {
            # create a subset of data examples that only belong to given
                  # digit class
            train.subset <- subset(train.data, train.data[,401] == label)
            pixels <- train.subset[,1:400]
            EM.info <- runEM(pixels, num.mix, phi, sigma, alpha)
            ten.EMs[[label + 1]] <- EM.info
      }
      return(ten.EMs)
}

EvaluateData <- function(ten.EMs, dev.data, num.mix) {
      # input: the EM information for the ten digit classes
      # output: an an accuracy on the development data

      # I implemented two different evaluation functions. One found the
      # marginal probability for each digit class by summing the joint
      # probabilites for each mixture component. However, I found better
      # results with the function implemented below. It also makes more
      # intuitive sense to me. It finds the maximum joint probability
      # across all mixtures and all digit classes, and classifies an image
      # to whichever class achieves this maximum.

      dev.pixels <- dev.data[,1:400]
      dev.labels <- dev.data[,401]
      correct <- 0
      for(i in 1:nrow(dev.pixels)) {
            # loop through each picture in the development data
            x <- dev.pixels[i,]
            #ten.probs <- rep(0, 10)
            most.likely = -Inf
            pred = -1
            for(k in 1:10) {
                  # loop through each digit class
                  prob.mat <- ten.EMs[[k]][[3]]
                  pi.vec <- ten.EMs[[k]][[4]]
                  prob.vec <- rep(0, num.mix)
                  for(m in 1:num.mix) {
                        prob <- dbinom(x, 1, prob.mat[,m])
                        m.prob <- sum(log(prob)) + log(pi.vec[m])
                        if(m.prob > most.likely) {
                              most.likely <- m.prob
                              pred <- k - 1 # get number between 0-9
                        }
                  }
```

```
            }
            if(pred == dev.labels[i]) {
                  correct <- correct + 1
            }
      }
      return(correct / nrow(dev.pixels))
}
```

```
getLogLik <- function(ten.EMs) {
      # input: the EM information across all ten digit classes
      # output: a vector of log likelihood information across the iterations

      min.iter <- Inf
      for(k in 1:10){
            # I need to find the minimum number of iterations until
                  # convergence was reached
            num.iter <- length(ten.EMs[[k]][[1]])
            if(num.iter < min.iter) {
                  min.iter <- num.iter
            }
      }

      log.liks <- rep(0, min.iter)
      for(i in 1:min.iter) {
            iter.sum <- 0
            for(j in 1:10) {
                  # sum the likelihoods from each digit class
                  iter.sum <- iter.sum + ten.EMs[[j]][[1]][i]
            }
            log.liks[i] <- iter.sum
      }
      return(log.liks)
}
```

One potential problem with this algorithm is that the accuracy will monotonically increase as the number of components increases. Theoretically, we could have 400 mixture components for each digit class, assigning each image to its own mixture. This would probably give us the best accuracy. And since our data set is standardized, such a model would probably do pretty well on the testing set too. In practice, the real problem with increasing the number of mixture components is that there are some components that are not assigned any images. We can see this by analyzing the weight matrix and finding any columns that do not have a value close to 1. In order to choose the optimal number of mixture components, I decided to choose the largest number of components such that every component had at least one image assigned to it. The function below implements that idea.

```
IsMixturesTooBig <- function(ten.EMs, num.mix, max.num) {
      # input: takes the weight matrices from the EM algorithm
      # output: checks to see that every column in the weight matrix
            # has at least one image assigned to it

      rv <- FALSE
      num.zero <- 0 # We want to end the iterations if there are
            # more than five mixture components with zero images
            # this achieves a balance between accuracy and sensibility
```

```
      for(k in 1:10) {
            for(m in 1:num.mix) {
                  if(max(ten.EMs[[k]][[2]][,m]) < 0.10) {
                        num.zero <- num.zero + 1
                        if(num.zero > max.num) {
                              return(TRUE)
                        }

                  }
            }
      }
      return(rv)
}
```

```
FinalFunction <- function(training.data, training.label, mixture.numbers,
                          num.trials, p, phi, sigma, alpha, max.num) {
      mix.accs <- c()
      all.log.liks <- c()
      for(num.mix in mixture.numbers) {
            avg.acc <- 0
            for(trial in 1:num.trials){
                  data.sets <- PrepareData(training.data, training.label, p)
                  train.data <- data.sets[[1]]
                  dev.data <- data.sets[[2]]
                  ten.EMs <- CreateTenMixtures(train.data, num.mix,
                                               phi, sigma, alpha)
                  if(IsMixturesTooBig(ten.EMs, num.mix, max.num)) {
                        print(paste("Optimal mixture components:", num.mix - 1))
                        return(list(mix.accs, all.log.liks, old.ten.EMs))
                  }
                  all.log.liks <- rbind(all.log.liks, getLogLik(ten.EMs))
                  acc <- EvaluateData(ten.EMs, dev.data, num.mix)
                  avg.acc <- avg.acc + acc
            }
            old.ten.EMs <- ten.EMs
      mix.accs <- append(mix.accs, avg.acc / num.trials)
      }
      return(list(mix.accs, all.log.liks, old.ten.EMs))
}
```

```
set.seed(66)
p <- 0.8
phi <- 2; sigma <- 2; alpha <- 2
mixture.numbers <- 1:10
num.trials <- 5
max.num <- 5 # how many mixtures with zero images do we want
final.info <- FinalFunction(training.data, training.label, mixture.numbers,
                            num.trials, p, phi, sigma, alpha, 5)
```

```
## [1] "Optimal mixture components: 5"
```

```
mix.accs <- final.info[[1]]
all.log.liks <- final.info[[2]]
ten.EMs <- final.info[[3]]
print(mix.accs)
```

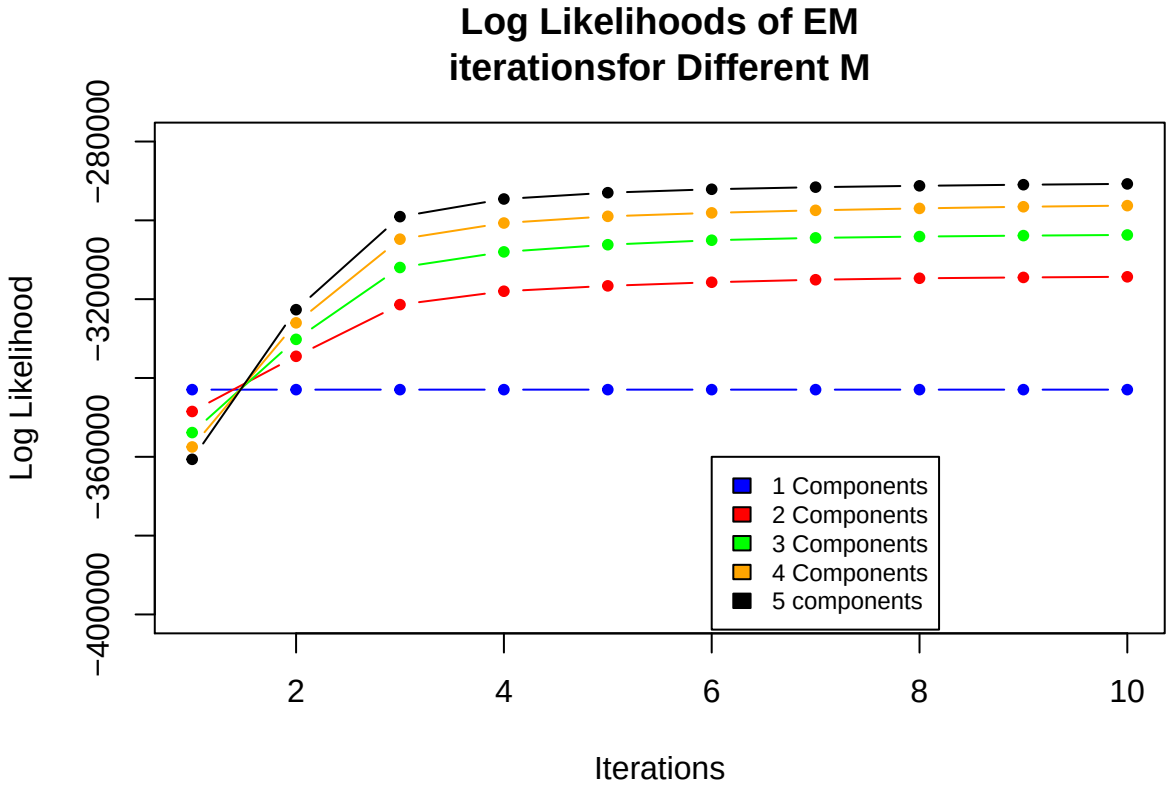
```
## [1] 0.8190 0.8550 0.8628 0.8730 0.8864
```

The numbers printed above are the average training accuracies from the development data set for each number of mixture components. Notice how they are monotonically increasing. This is to be expected as adding more mixtures should never decrease the classification accuracy. However, we stopped our algorithm when we found more than five mixture components across all of the digits that had zero images assigned to it. This resulted in a model that uses 5 mixture components. Ideally, we would use a different number of components for each digit class, but that is beyond the scope of this project. Below we compare the development accuracy for a model with 5 mixture components to one with 15.

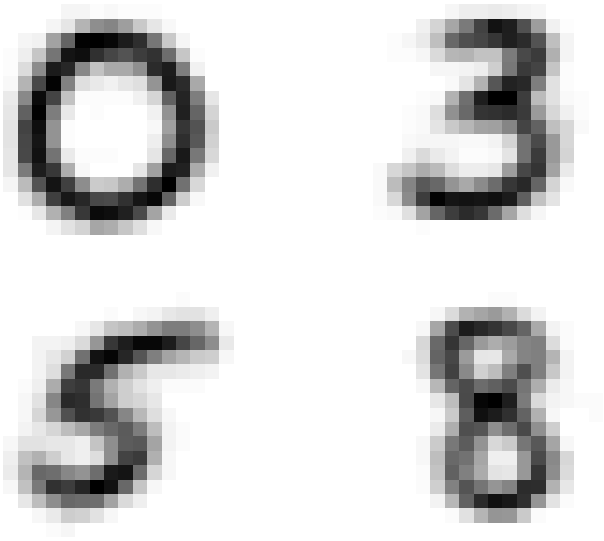
```
final.info <- FinalFunction(training.data, training.label, 15,
                            num.trials, p, phi, sigma, alpha, 150)
mix.accs <- final.info[[1]]
print(mix.accs)
```

```
## [1] 0.881
```

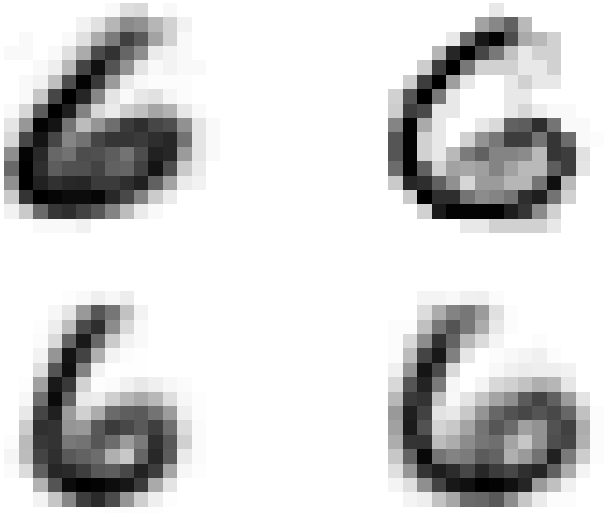
The development accuracy is actually worse with this many mixture components, so we are satisfied in our choice of using 5 mixture components.



Here, we plot the log likelihood function for three different mixture models. Notice that the log likelihood is maximized when we use more mixture components. Also, there is a big increase in the log likelihood after the first few iterations. After that, the algorithm converges.



Above, we print out some of the resulting pixel probability vectors from the model with 5 mixture components. The probability vectors have shaped themselves to represent the digits that they attempt to classify.



These pictures represent four of the five components for the digit class 6. We can see that the EM algorithm was able to find the different ways that people draw the number 6.

```
# Here we use our 15 component mixture model to evaluate the unseen test
      # data.
best.M <- 5
test.data.labels <- cbind(test.data, test.label)
test.acc = EvaluateData(ten.EMs, test.data.labels, best.M)
print(test.acc)
```

```
## [1] 0.8831
```