SVM

Michael Frasco

RESULTS: For full SVM I achived an accuracy around 90% on the unseen testing data. I used a regularization parameter of 0.1.

RESULTS: With stochastic gradient descent, I achieved an accuracy of about 87% by performing 25 iterations over the data. I found that a smaller regularization parameter of 0.01 led to the best cross-validated results. I also used a smaller value for the step size by reducing the step after each observation.

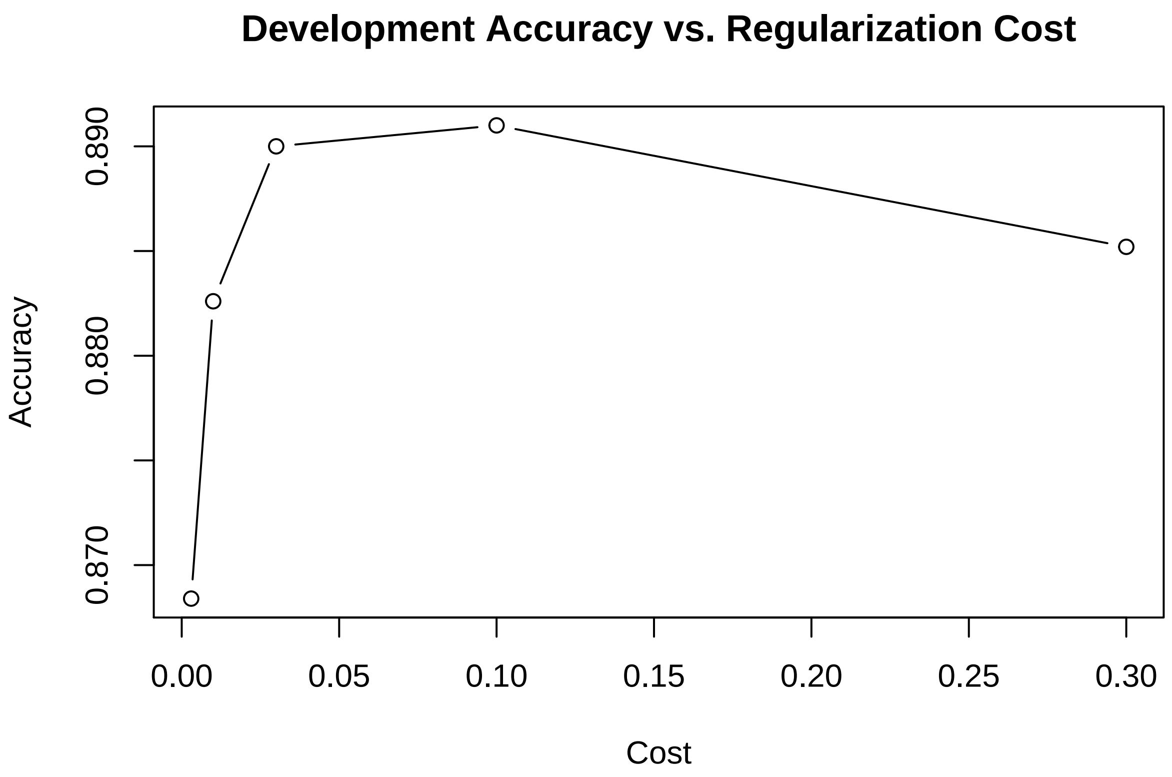
```
PrepareData <- function(training.data, training.label, p) {
      # input: training data, training labels, and the proportion of the
            # data that should be used to train
      # output: a list of the training and development set

      training.data = cbind(training.data, training.label)
      for(i in 0:9) {
            # Create binary indicators for each digit class
            # to be used when implementing gradient descent
            y.vec <- ifelse(training.data[,401] == i, 1, -1)
            training.data = cbind(training.data, y.vec)
      }
      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))
}
```

Full SVM

```
p <- 0.8
cost.values <- c(0.003, 0.01, 0.03, 0.1, 0.3) # various cost values to try
acc.vec <- c()
for(cst in cost.values) {
      accs <- rep(0, 5)
      for(trial in 1:5) { # repeat each cost value five times
            data.sets <- PrepareData(training.data, training.label, 0.8)
            train.data <- data.sets[[1]]
            dev.data <- data.sets[[2]]
            colnames(train.data)[401] <- c("digit")

            svm.model <- svm(digit ~ ., data = train.data[,1:401],
                  kernel = "linear", cost = cst, scale = FALSE, type="C")
            # scale equals false to prevent issues
            # with pixels that are always 0
            svm.pred <- predict(svm.model, dev.data[,-401])
            accs[trial] <- mean(svm.pred==dev.data[,401])
      }
      acc.vec <- append(acc.vec, mean(accs))
}
print(acc.vec)
```

```
## [1] 0.8684 0.8826 0.8900 0.8910 0.8852
```

We find that a regularization parameter of 0.1 provides the best cross validation accuracy. We searched over possible lambda values. Searching for a better lambda in the intermediary values did not result in a significant change in the accuracy of the prediction. The graph below shows how the average error rate on the validation set changed with lambda.



```r
# Now we generate a model with the best value for lambda on the test data
best.cost <- cost.values[which.max(acc.vec)]
data.sets <- PrepareData(training.data, training.label, 0.8)
train.data <- data.sets[[1]]
colnames(train.data)[401] <- c("digit")
test.data.labels <- cbind(test.data, test.label)

svm.model <- svm(digit ~ ., data = train.data[,1:401],
     kernel = "linear", cost = best.cost, scale = FALSE, type='C')
svm.pred <- predict(svm.model, test.data.labels[,-401])
test.acc <- mean(svm.pred == test.data.labels[,401])
print(test.acc)
```

```
## [1] 0.8908
```

Above is the prediction accuracy on the 10,000 unseen test images.

Stochastic gradient descent

Below, I implement stochastic gradient descent. I experimented a lot with the implementation of this algoritm. I tried various values for the regularization parameter lambda, for the intial step size, and for the update process of the step size. I found that dividing the step size by the number of observations seen provided the best results. I also considered dividing by the square root of the number of observations seen and dividing by the number of passes made over the data. The logic behind dividing after each observation instead of after each pass is that the update of the weights can be smaller so that the algorithm converges exactly on the right values. Also important was the balance between the regularization parameter and the step size. Since the regularization parameter was so small, the step size had to adjust accordingly.

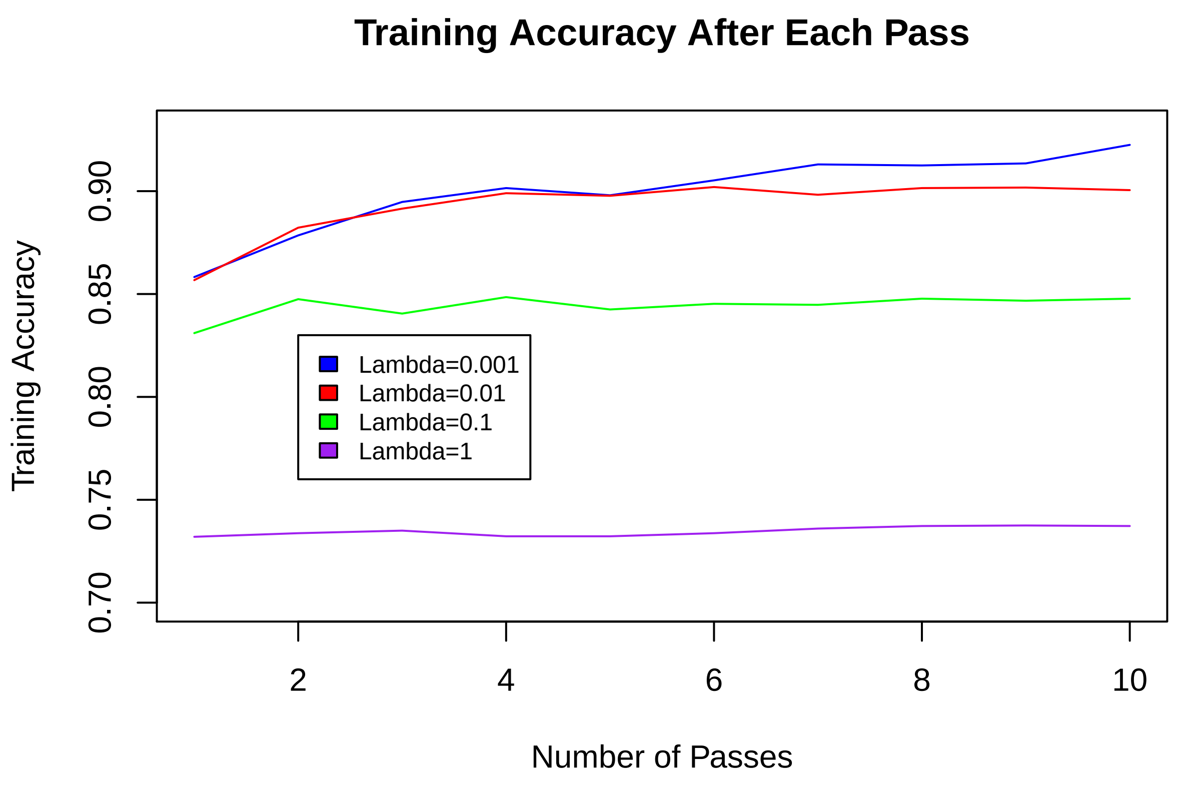
Also, I added a column of “1”s to the training data and the testing data to serve as the intercept term for each classifier. The idea is that numbers with a large amount of pixels activated (e.g. “5”, “8”) will have different intercept than numbers with a small amount of pixels (e.g. “1”)

```
MakePass <- function(lambda, weights, train.data, pass.num) {
      # input: the regularization parameter lambda, the weights for
            # each class, the training data, the number of times the
            # data has already been passed over
      # output: an updated list of weights

      for(k in 1:10) {
            # loop over each class
            W.vec <- weights[[k]]
            step.size.initial <- (1/2) * (1/lambda)
            # shuffle the data
            shuffled.data <- train.data[sample(nrow(train.data)),]
            shuffled.Y <- shuffled.data[,402 + k]
            t <- (pass.num - 1) * 4000 + 1

            for (n in 1:nrow(train.data)) {
                  # reduce the step size every observation
                  step.size <- step.size.initial / t
                  yx <- as.numeric(shuffled.Y[n] * shuffled.data[n, 1:401])
                  pred <- as.numeric(t(yx) %*% W.vec)
                  if(pred > 1) {
                        # we made the correct prediction
                        W.vec <- (1 - step.size * lambda) * W.vec
                  } else {
                        # incorrect prediction
                        W.vec <- (1 - step.size * lambda) * W.vec + step.size * yx
                  }
                  t <- t + 1
            }
            weights[[k]] <- W.vec
      }
      return(weights)
}

EvaluateWeights <- function(weights, data) {
      # input: a list of weights for all ten digit classes
      # output: the error rate

      preds <- rep(0, nrow(data))
      for(n in 1:nrow(data)) {
            maxVal <- -Inf
            maxK <- -1
            x <- data[n, 1:401]
            for(k in 1:10) {
                  kVal <- as.numeric(t(weights[[k]]) %*% x)
                  if(kVal > maxVal) {
                        maxVal <- kVal
                        maxK <- k
                  }
            }
            preds[n] = maxK - 1
      }
      acc <- mean(preds == data[,402])
      return(acc)
}

RunGradDescent <- function(lambda, train.data, num.passes) {
      # input: a value of lambda, the training data, the number of passes
            # to make over the data
      # output: the final weights and the error on the training data
            # after each iteration

      weights <- list()
      for(k in 1:10) {
            weights[[k]] <- rep(0, 401)
      }
      train.accs <- c()
      for(s in 1:num.passes) {
            weights <- MakePass(lambda, weights, train.data, s)
            train.accs <- append(train.accs, EvaluateWeights(weights, train.data))
      }
      return(list(weights, train.accs))
}
```

```
lambdaVal <- c(0.001, 0.01, 0.1, 1) # some initial values of lambda to test
train.accs <- c()
dev.accs <- c()
num.passes <- 10
p = 0.8
for(lambda in lambdaVal) {
      data.sets <- PrepareData(training.data, training.label, p)
      train.data <- as.matrix(data.sets[[1]])
      train.data <- cbind(rep(1, nrow(train.data)), train.data)
      dev.data <- as.matrix(data.sets[[2]])
      dev.data <- cbind(rep(1, nrow(dev.data)), dev.data)

      grad.info <- RunGradDescent(lambda, train.data, num.passes)
      weights <- grad.info[[1]]
      train.accs <- rbind(train.accs, grad.info[[2]])
      dev.accs <- append(dev.accs, EvaluateWeights(weights, dev.data))
}
print(dev.accs)
```

```
## [1] 0.852 0.873 0.839 0.739
```

Above is the accuracy on the development data for each value of lambda. Below is a plot of the accuracy on the training data after each pass over the data.



After that intial pass through the data, we find that a regularization parameter between 0.001 and 0.01 is probably going to be the best. We focus our search on this value and increase the number of passes over the data.

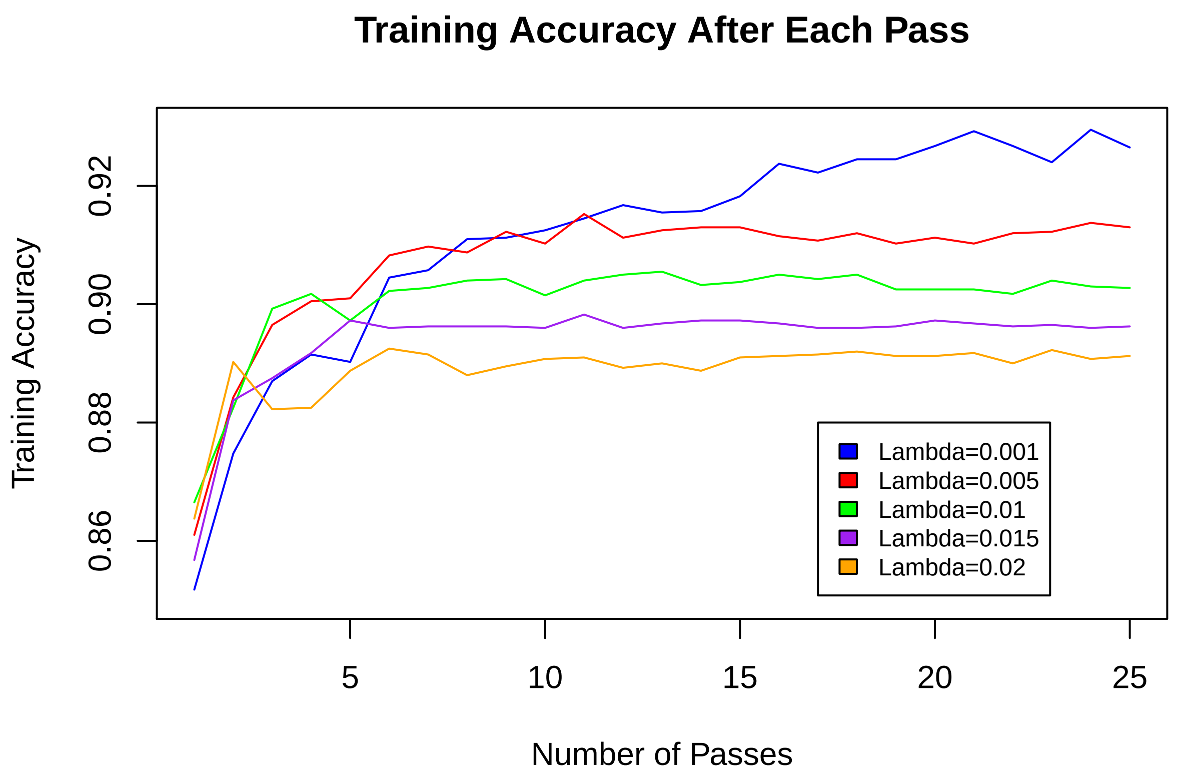
```
lambdaVal <- c(0.001, 0.005, 0.01, 0.015, 0.02)
train.accs <- c()
dev.accs <- c()
num.passes <- 25
p = 0.8
for(lambda in lambdaVal) {
      data.sets <- PrepareData(training.data, training.label, p)
      train.data <- as.matrix(data.sets[[1]])
      train.data <- cbind(rep(1, nrow(train.data)), train.data)
      dev.data <- as.matrix(data.sets[[2]])
      dev.data <- cbind(rep(1, nrow(dev.data)), dev.data)

      grad.info <- RunGradDescent(lambda, train.data, num.passes)
      weights <- grad.info[[1]]
      train.accs <- rbind(train.accs, grad.info[[2]])
      dev.accs <- append(dev.accs, EvaluateWeights(weights, dev.data))
}
print(dev.accs)
```

```
## [1] 0.878 0.890 0.874 0.871 0.876
```

```
print(lambdaVal[which.max(dev.accs)])
```

```
## [1] 0.005
```

Here, we see the value of lambda that maximizes the accuracy on the development set.



In the plot above, we see evidence of overfitting. The smallest value of lambda performs well on the training data, but it does not do as well on the development data.

```
num.passes <- 25
best.lambda <- lambdaVal[which.max(dev.accs)]
data.sets <- PrepareData(training.data, training.label, p)
train.data <- as.matrix(data.sets[[1]])
train.data <- cbind(rep(1, nrow(train.data)), train.data)
dev.data <- as.matrix(data.sets[[2]])
dev.data <- cbind(rep(1, nrow(dev.data)), dev.data)

grad.info <- RunGradDescent(best.lambda, train.data, num.passes)
weights <- grad.info[[1]]
test.data.label <- cbind(rep(1, nrow(test.data)), test.data, test.label)
test.acc <- EvaluateWeights(weights, test.data.label)
print(test.acc)
```

```
## [1] 0.8721
```

The accuracy on the unseen testing data is printed above.