# CSE 6242 Assignment 3

*Vincent La (Georgia Tech ID - vla6)*

*October 21, 2017*

## Question 1: Data Preprocessing

```
setwd("~/git/GeorgiaTech/cse6242/assignment_3")
train = read.csv('./mnist/mnist_train.csv', header=FALSE)
test = read.csv('./mnist/mnist_test.csv', header=FALSE)

last_row = nrow(train)

# Hacky approach to do preprocessing; I'm sure there's a more efficient way
train_0_1_cols = c()
train_3_5_cols = c()
for (i in 1:ncol(train)){
  if (train[last_row, i] == 0 | train[last_row, i] == 1){
    train_0_1_cols = c(train_0_1_cols, colnames(train)[i])
  } else if (train[last_row, i] == 3 | train[last_row, i] == 5){
    train_3_5_cols = c(train_3_5_cols, colnames(train[i]))
  }
}

test_0_1_cols = c()
test_3_5_cols = c()
for (i in 1:ncol(test)){
  if (test[last_row, i] == 0 | test[last_row, i] == 1){
    test_0_1_cols = c(test_0_1_cols, colnames(test)[i])
  } else if (test[last_row, i] == 3 | test[last_row, i] == 5){
    test_3_5_cols = c(test_3_5_cols, colnames(test[i]))
  }
}

train_0_1 = train[train_0_1_cols]
train_3_5 = train[train_3_5_cols]
test_0_1 = test[test_0_1_cols]
test_3_5 = test[test_3_5_cols]

print(dim(train_0_1))
```

```
## [1]   785 12665
```

```
print(dim(train_3_5))
```

```
## [1]   785 11552
```

```
print(dim(test_0_1))
```

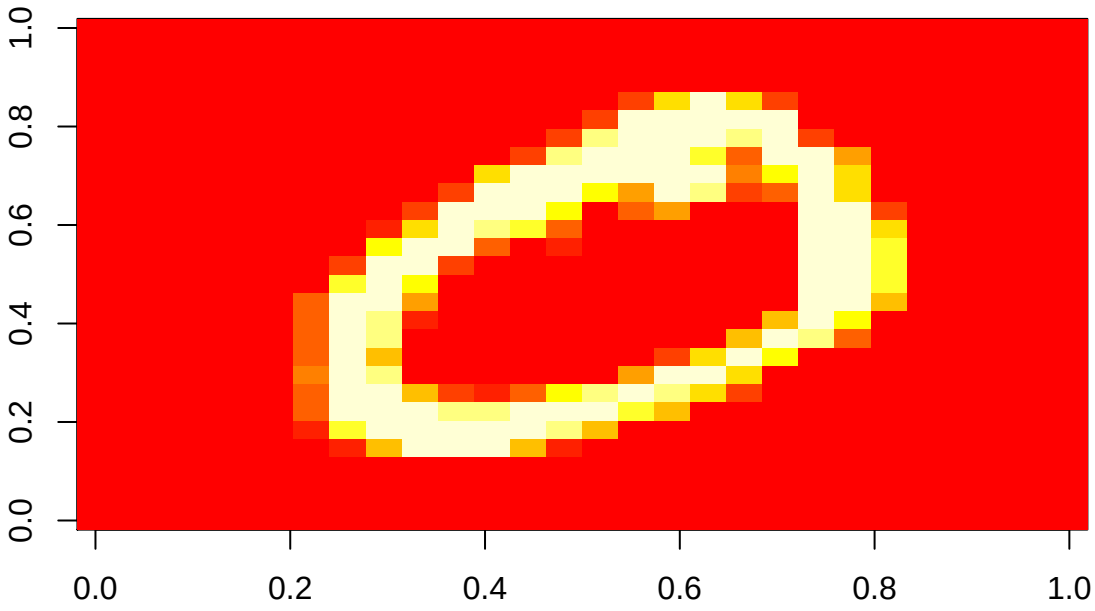
```
## [1]  785 2115
```

```
print(dim(test_3_5))
```

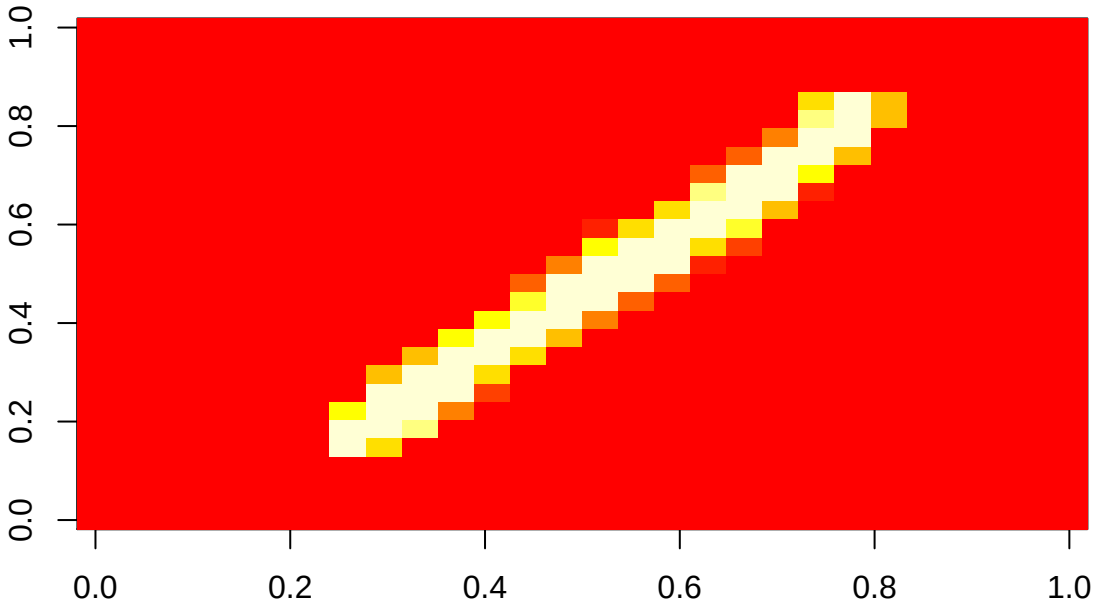
```
## [1]  785 1902
```

```
# Removing true image labels
train_labels_0_1 = train_0_1[last_row, ]
test_labels_0_1 = test_0_1[last_row, ]
train_labels_3_5 = train_3_5[last_row, ]
test_labels_3_5 = test_3_5[last_row, ]

# Deleting the image labels from DF
train_0_1 = train_0_1[-c(last_row), ]
test_0_1 = test_0_1[-c(last_row), ]
train_3_5 = train_3_5[-c(last_row), ]
test_3_5 = test_3_5[-c(last_row), ]
```

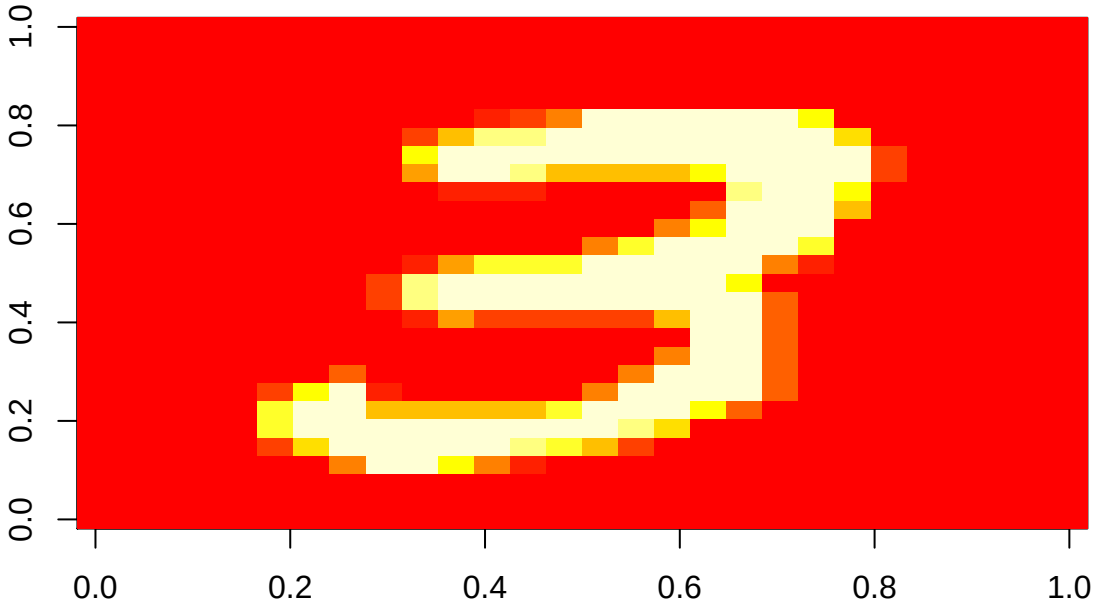
```
ex_0_matrix = matrix(train_0_1[, 'V1'], nrow=28, ncol=28)
ex_1_matrix = matrix(train_0_1[, 'V12664'], nrow=28, ncol=28)
ex_3_matrix = matrix(train_3_5[, 'V12666'], nrow=28, ncol=28)
ex_5_matrix = matrix(train_3_5[, 'V24217'], nrow=28, ncol=28)

# Taken from https://www.r-bloggers.com/creating-an-image-of-a-matrix-in-r-using-image/
rotate <- function(x) t(apply(x, 2, rev))
image(rotate(ex_0_matrix))
```



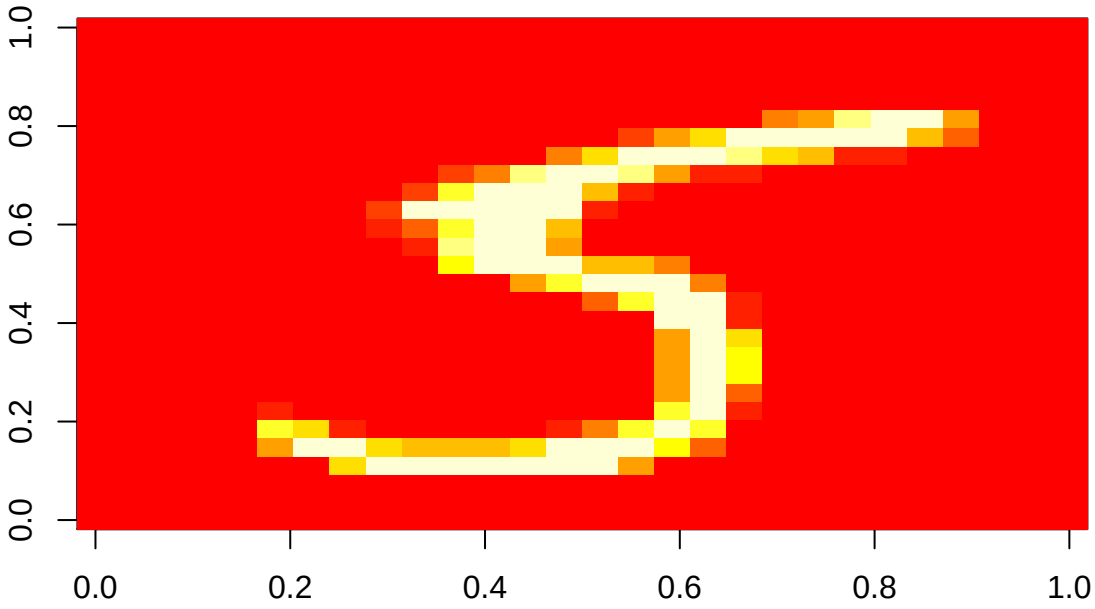

```
image(rotate(ex_1_matrix))
```




```
image(rotate(ex_3_matrix))
```




```
image(rotate(ex_5_matrix))
```




## Question 2: Theory

### Part a: Write down the formula for the loss function used in Logistic Regression, the expression that you want to minimize: L($\theta$)

Taken from the lecture "MLE and Iterative Optimization":

$$\hat{\theta}_{MLE} = argmin_\theta \sum_{i=1}^{n} log(1 + e^{y^{(i)} * <\theta, x^{(i)}>})$$

Thus, the loss function is

$$L(\theta) = \sum_{i=1}^{n} log(1 + e^{y^{(i)} * <\theta, x^{(i)}>})$$

where $y^{(i)} = 1$ or $y^{(i)} = -1$.

### Part b: Derive the gradient of the loss function with respect to model parameters: $\frac{dL(\theta)}{d\theta}$ or $\frac{\partial L(\theta)}{\partial \theta_j}$.

$$\frac{\partial L(\theta)}{\partial \theta_j} = \frac{\partial \sum_{i=1}^{n} log(1 + e^{y^{(i)} * <\theta, x^{(i)}>})}{\theta_j}$$

Furthermore, we know that $\frac{\partial}{\partial x} log(x) = \frac{1}{x}$. Also, we can use the chain rule here to complete the derivative.

$$\frac{\partial L(\theta)}{\partial \theta_j} = \sum_{i=1}^{n} \frac{1}{1 + e^{y^{(i)} * <\theta, x^{(i)}>}} * \frac{\partial}{\partial \theta_j} e^{y^{(i)} * <\theta, x^{(i)}>}$$

$$\frac{\partial L(\theta)}{\partial \theta_j} = \sum_{i=1}^{n} \frac{e^{y^{(i)} * <\theta, x^{(i)}>}}{1 + e^{y^{(i)} * <\theta, x^{(i)}>}} * \frac{\partial}{\partial \theta_j}(y^{(i)} * <\theta, x^{(i)}>)$$

$$\frac{\partial L(\theta)}{\partial \theta_j} = \sum_{i=1}^{n} \frac{(e^{y^{(i)} * <\theta, x^{(i)}>}) * y^{(i)} x_j^{(i)}}{1 + e^{y^{(i)} * <\theta, x^{(i)}>}}$$

## Part c: Based on this gradient, express the Stochastic Gradient Descent (SGD) update rule that uses a single sample $< x^{(i)}, y^{(i)} >$ at a time.

Stochastic Gradient Descent (SGD) can be used when your data is very big. The steps are:

1. Initialize the dimensions of $\theta$ vector to random values.
2. Pick one labeled data vector $(x^{(i)}, y^{(i)})$ randomly, and update for each $j = 1, ..., d : \theta_i \leftarrow \theta_j - \alpha \frac{\partial log(1+exp(y^{(i)} <\theta, x^{(i)}>))}{\partial \theta_j}$
3. Repeat step (2) until the updates of the dimensions of $\theta$ become too small.

So basically substituting in the expression for $\frac{\partial}{\partial \theta_j} log(1 + exp(y^{(i)} < \theta, x^{(i)} >))$ that we found previously, we get:

$$\frac{\partial}{\partial \theta_j} log(1 + exp(y^{(i)} < \theta, x^{(i)} >)) = \frac{(e^{y^{(i)} * <\theta, x^{(i)}>}) * y^{(i)} x_j^{(i)}}{1 + e^{y^{(i)} * <\theta, x^{(i)}>}}$$

Thus, the update rule becomes,

for each $j = 1, ..., d : \theta_i \leftarrow \theta_j - \alpha * \frac{(e^{y^{(i)} * <\theta, x^{(i)}>}) * y^{(i)} x_j^{(i)}}{1 + e^{y^{(i)} * <\theta, x^{(i)}>}}$

## Part d: Write pseudocode for training a model using Logistic Regression and SGD.

1. for (j from 1, . . . , d), initialize $\theta_j$ to random values. (Initialize the dimensions of $\theta$ vector to random values.)
2. Pick one labeled data vector randomly, call it $(x^{(i)}, y^{(i)})$
3. for (j from 1, . . . , d) set $\theta_i$ equal to $\theta_j - \alpha * \frac{(e^{y^{(i)} * <\theta, x^{(i)}>}) * y^{(i)} x_j^{(i)}}{1 + e^{y^{(i)} * <\theta, x^{(i)}>}}$, where $\alpha$ is some step size, decaying as the gradient descent iterations increase.
4. Repeat step (3) until the updates of the dimensions of $\theta$ become too small. To be more specific, compute absolute sum of the updates to each $\theta_j$ at each iteration, If the absolute sum of the updates is small, say 0.000001, then terminate.

## Part e: Estimate the number of operations per epoch of SGD, where an epoch is one complete iteration through all the training samples. Express this is Big-O notation, in terms of the number of samples (n) and the dimensionality of each sample (d).

In each epoch, we have $n$ samples, and thus $n$ iterations. We also have to update $d$ dimensions of the $\theta$ vector. Thus, the complexity is linear with respect to $n$ and $d$ or $O(n * d)$.

## Section 3: Implementation

```
# Processing the Data
## Changing the labels to vectors. Also map values 0 and 3 to -1; 1 and 5 to 1
train_labels_0_1 = as.vector(train_labels_0_1, mode='numeric')
train_labels_0_1_orig = train_labels_0_1
train_labels_0_1 = mapvalues(train_labels_0_1, from=c(0, 3), to=c(-1, -1), warn_missing=FALSE)

train_labels_3_5 = as.vector(train_labels_3_5, mode='numeric')
train_labels_3_5_orig = train_labels_3_5
train_labels_3_5 = mapvalues(train_labels_3_5, from=c(0, 3), to=c(-1, -1), warn_missing=FALSE)
train_labels_3_5 = mapvalues(train_labels_3_5, from=c(5), to=c(1), warn_missing=FALSE)

test_labels_0_1 = as.vector(test_labels_0_1, mode='numeric')
test_labels_0_1_orig = test_labels_0_1
test_labels_0_1 = mapvalues(test_labels_0_1, from=c(0, 3), to=c(-1, -1), warn_missing=FALSE)

test_labels_3_5 = as.vector(test_labels_3_5, mode='numeric')
test_labels_3_5_orig = test_labels_3_5
test_labels_3_5 = mapvalues(test_labels_3_5, from=c(0, 3), to=c(-1, -1), warn_missing=FALSE)
test_labels_3_5 = mapvalues(test_labels_3_5, from=c(5), to=c(1), warn_missing=FALSE)

## Adding the bias term
train_0_1_w_bias = rbind(train_0_1, rep(1, ncol(train_0_1)))
train_3_5_w_bias = rbind(train_3_5, rep(1, ncol(train_3_5)))
test_0_1_w_bias = rbind(test_0_1, rep(1, ncol(test_0_1)))
test_3_5_w_bias = rbind(test_3_5, rep(1, ncol(test_3_5)))

train = function(data, labels, alpha){
  # Train a Logistic Regression model
  # Keyword args:
  #   data: matrix or dataframe containing the input features (x^(i))
  #   labels: vector containing the corresponding labels (y^(i))
  #   alpha: learning rate hyperparameter
  # Returns:
  #   theta: vector of model parameters (theta) that minimizes the logistic regression loss L(theta)

  d = nrow(data)
  n = ncol(data)
  epochs = 100  # Total number of epochs, arbitrarily set to 100 for now

  # 1. Initialize theta_j to random values
  theta = runif(d, -0.5, 0.5)
  e = 1

  while(e <= epochs){
    # For each epoch shuffle the data
    shuffled_idx = sample(n, replace=FALSE)
    number_iterations = 0

    prev_theta = theta
    for(i in shuffled_idx){
      # 2. Pick one labeled data vector. This works because we earlier shuffled the data
```

```
      x_i = data[, i]
      y_i = labels[i]

      # 3. Apply update rule
      numerator = x_i * y_i
      denominator = 1 + exp(sum(theta * x_i) * y_i)
      update = numerator / denominator
      theta = theta + alpha * update

      number_iterations = number_iterations + 1
      if (sum(abs(alpha * (numerator/denominator))) < 0.000001){
        return(theta)
      }
    }
    e = e + 1
  }

  # If algorithm has not yet converged after epoch limit, return theta anyway
  print("i never converged!")
  return(theta)
}

predict = function(theta, data){
  # Train a Logistic Regression model
  # Keyword args:
  #   theta: vector of model parameters (theta), as returned by train()
  #   data: matrix or dataframe containing the input features (x^(i))
  # Returns:
  #   labels: vector containing predicted labels (\hat{y}^(i))
  data = as.matrix(data)
  labels = sign(theta %*% data)
  return(as.vector(labels, mode='numeric'))
}

# Setting alpha constant
alpha = 0.01

theta_0_1 = train(data=train_0_1_w_bias, labels=train_labels_0_1, alpha=alpha)
training_pred_0_1 = predict(theta_0_1, data=train_0_1_w_bias)

theta_3_5 = train(data=train_3_5_w_bias, labels=train_labels_3_5, alpha=alpha)
training_pred_3_5 = predict(theta_3_5, data=train_3_5_w_bias)

table(training_pred_0_1, train_labels_0_1)
```

```
##                  train_labels_0_1
## training_pred_0_1   -1    1
##                -1 5914   41
##                1     9 6701
```

```
table(training_pred_3_5, train_labels_3_5)
```

```
##                  train_labels_3_5
## training_pred_3_5   -1    1
```

```
##                -1 5998  637
##                1   133 4784
```

### Implementation Notes

Note that to implement this algorithm, I added a row (basically another feature) for the bias term (a vector one 1's). This allows it so that the hyperplane classifier does not have to necessarily pass through the origin. Furthermore, I mapped the values (0,1) to (-1, +1), respectively. In addition, I mapped the values (3,5) to (-1, +1) as well, respectively.

### Initialization Method

I initialized $\theta$ with uniform random values between [-0.5, 0.5]

### Convergence Criteria

I calculated the absolute sum of the updates to each term of the $\theta$ vector. When the absolute sum of the update was less than 0.000001, I considered it convereged and stopped.

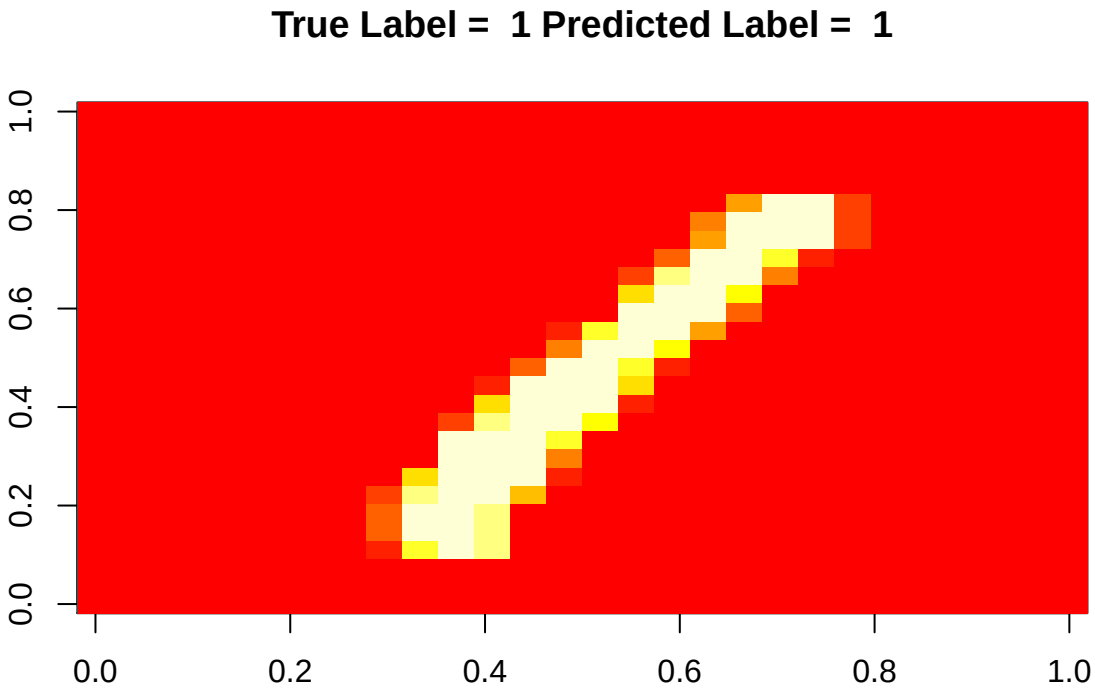
### Modifications

One minor modification is that I do shuffle the data for each epoch. Otherwise, mostly the same as the update rule discussed in Problem (2).

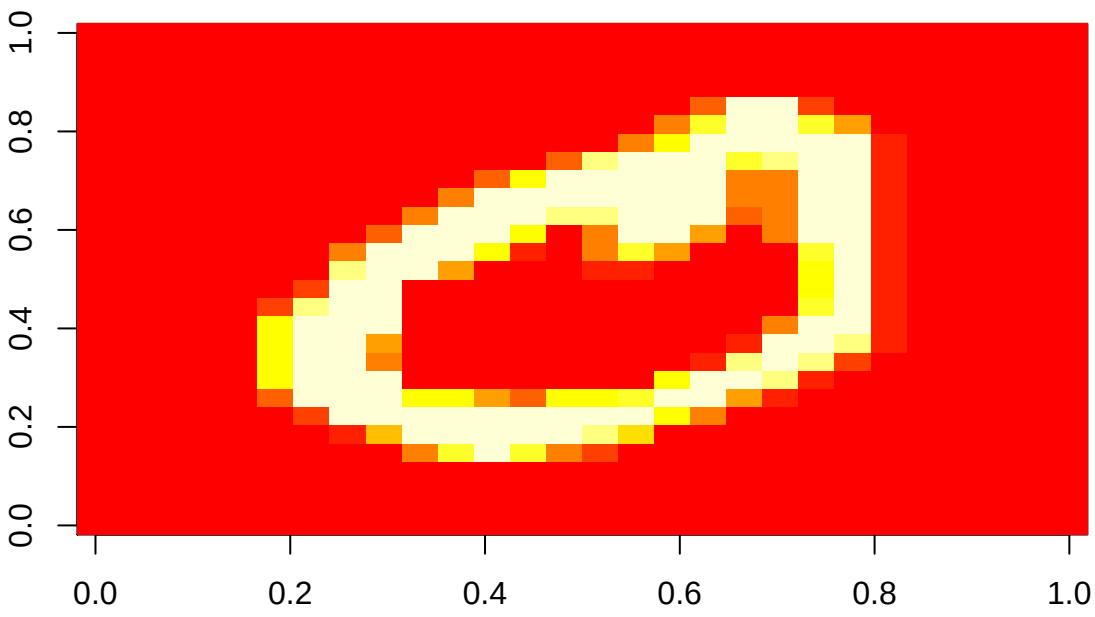
### Visualizations

Below we show the visualizations of the numbers along with the true label and predicted label. Recall that we transformed the labels such that (0, 3) got mapped to (-1) and (1, 5) got mapped to (+1).

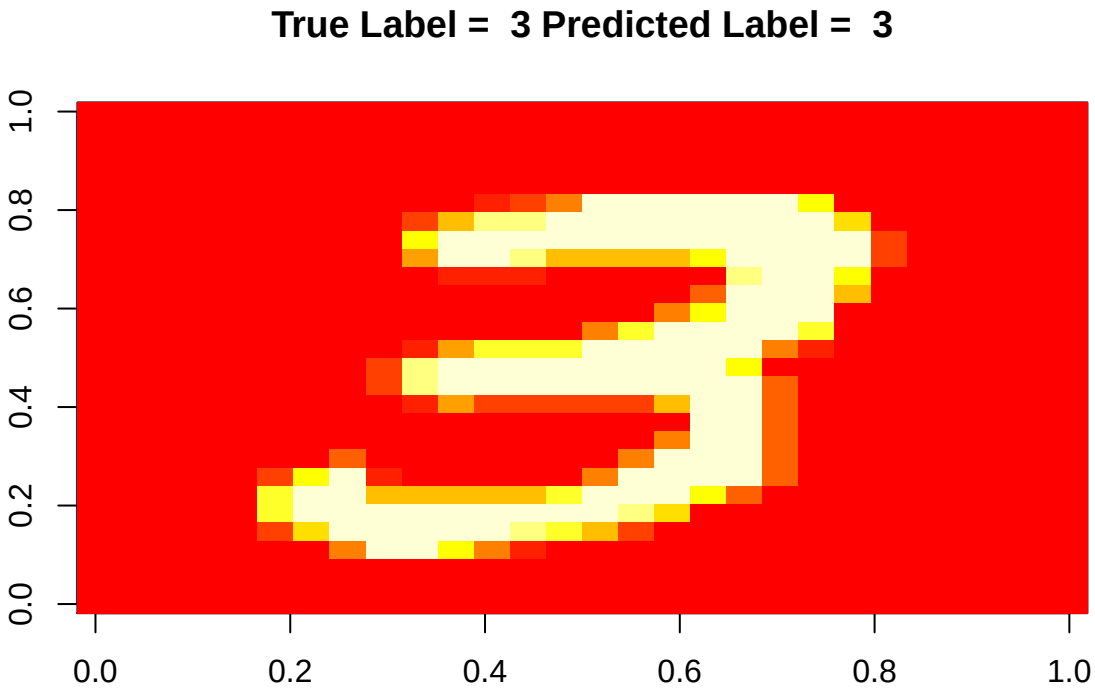
```
# First, we will visualize 2 correct predictions for the 0/1 and 3/5 training sets (4 total visualizati
correct_0_1_a = which(training_pred_0_1 == train_labels_0_1 & training_pred_0_1 == 1)[1]
correct_0_1_b = which(training_pred_0_1 == train_labels_0_1 & training_pred_0_1 == -1)[2]
correct_3_5_a = which(training_pred_3_5 == train_labels_3_5 & training_pred_3_5 == -1)[1]
correct_3_5_b = which(training_pred_3_5 == train_labels_3_5 & training_pred_3_5 == 1)[2]

correct_0_1_a_matrix = matrix(train_0_1[, correct_0_1_a], nrow=28, ncol=28)
correct_0_1_b_matrix = matrix(train_0_1[, correct_0_1_b], nrow=28, ncol=28)
correct_3_5_a_matrix = matrix(train_3_5[, correct_3_5_a], nrow=28, ncol=28)
correct_3_5_b_matrix = matrix(train_3_5[, correct_3_5_b], nrow=28, ncol=28)

image(rotate(correct_0_1_a_matrix), main=paste('True Label = ', 1, 'Predicted Label = ', 1))
```



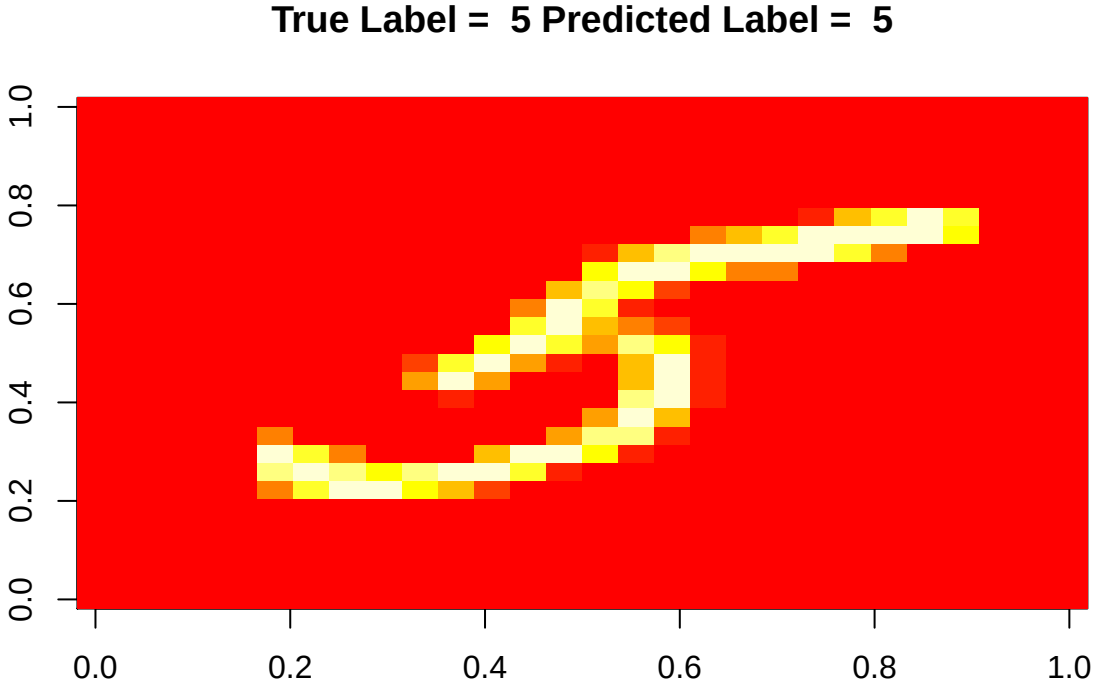

```
image(rotate(correct_0_1_b_matrix), main=paste('True Label = ', 0, 'Predicted Label = ', 0))
```




```
image(rotate(correct_3_5_a_matrix), main=paste('True Label = ', 3, 'Predicted Label = ', 3))
```

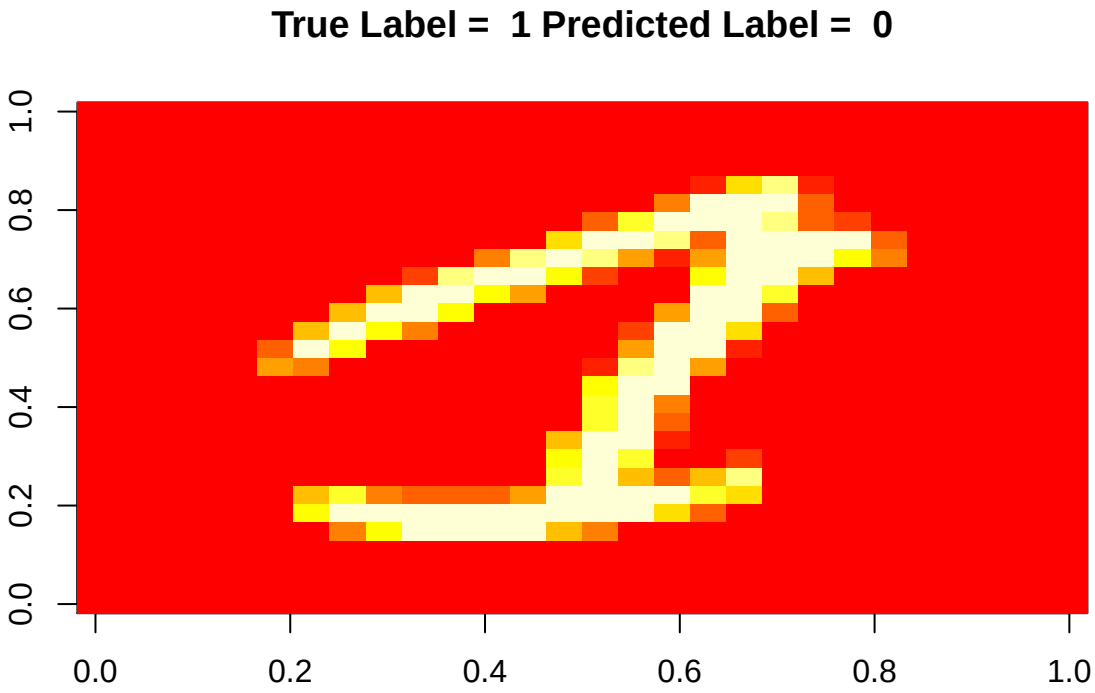



```
image(rotate(correct_3_5_b_matrix), main=paste('True Label = ', 5, 'Predicted Label = ', 5))
```

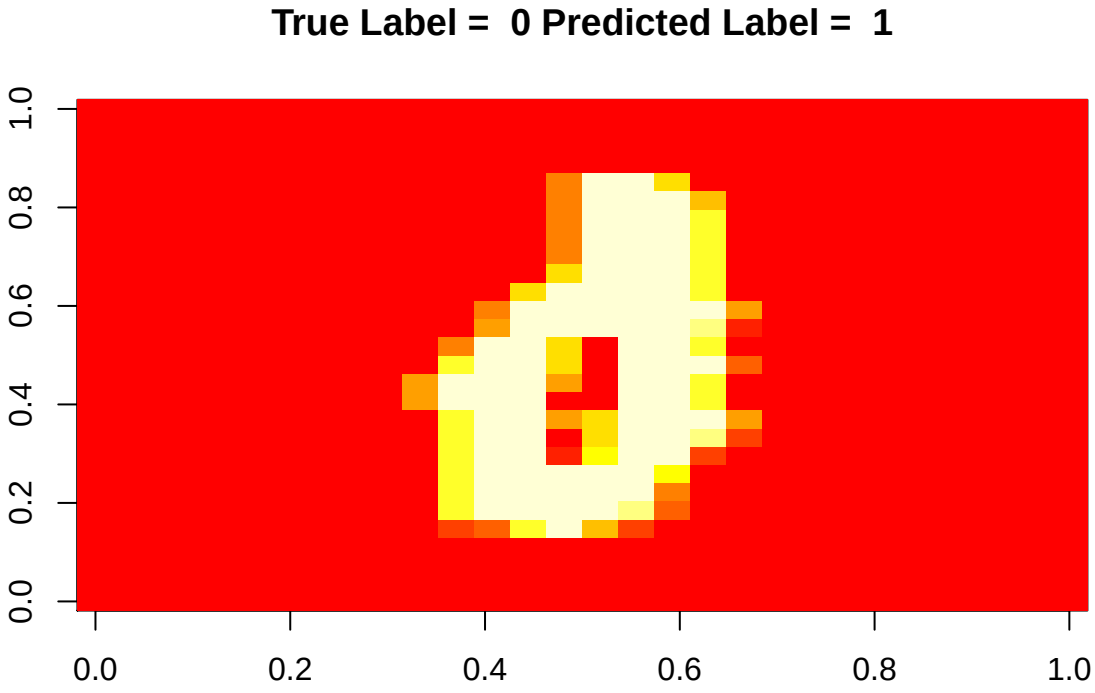



```
# Second, we will visualize 2 incorrect predictions for the 0/1 and 3/5 training sets (4 total visualiz
incorrect_0_1_a = which(training_pred_0_1 != train_labels_0_1 & training_pred_0_1 == -1)[1]
incorrect_0_1_b = which(training_pred_0_1 != train_labels_0_1 & training_pred_0_1 == 1)[2]
incorrect_3_5_a = which(training_pred_3_5 != train_labels_3_5 & training_pred_3_5 == -1)[1]
incorrect_3_5_b = which(training_pred_3_5 != train_labels_3_5 & training_pred_3_5 == 1)[2]

incorrect_0_1_a_matrix = matrix(train_0_1[, incorrect_0_1_a], nrow=28, ncol=28)
incorrect_0_1_b_matrix = matrix(train_0_1[, incorrect_0_1_b], nrow=28, ncol=28)
```

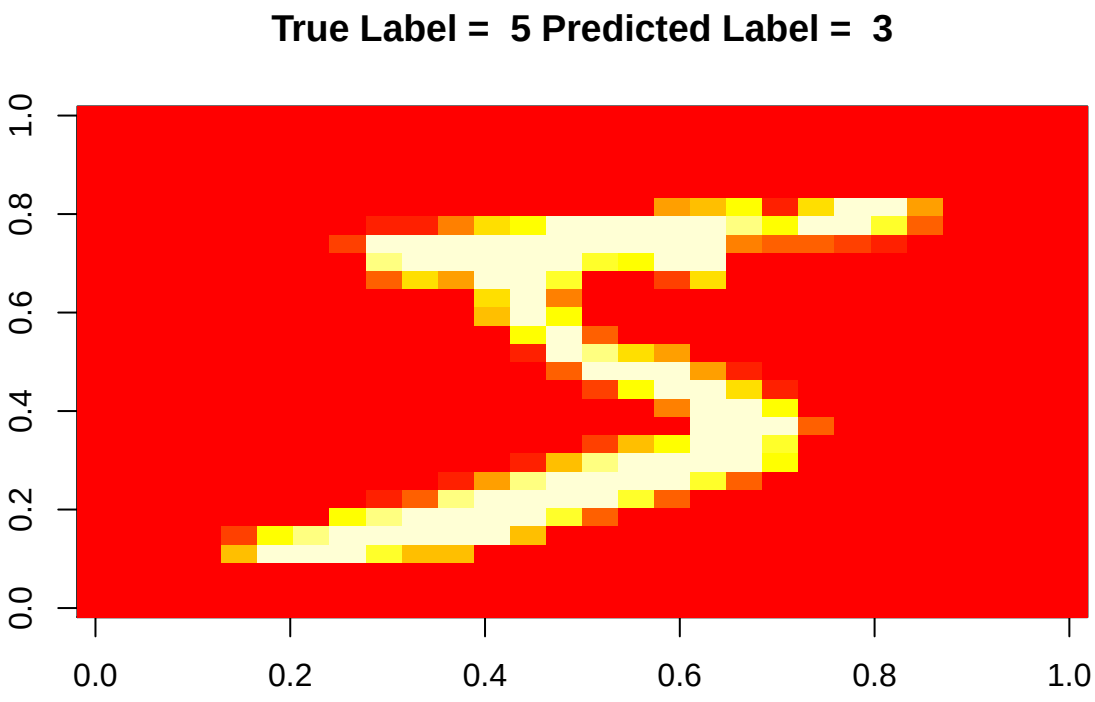
```
incorrect_3_5_a_matrix = matrix(train_3_5[, incorrect_3_5_a], nrow=28, ncol=28)
incorrect_3_5_b_matrix = matrix(train_3_5[, incorrect_3_5_b], nrow=28, ncol=28)

image(rotate(incorrect_0_1_a_matrix), main=paste('True Label = ', 1, 'Predicted Label = ', 0))
```



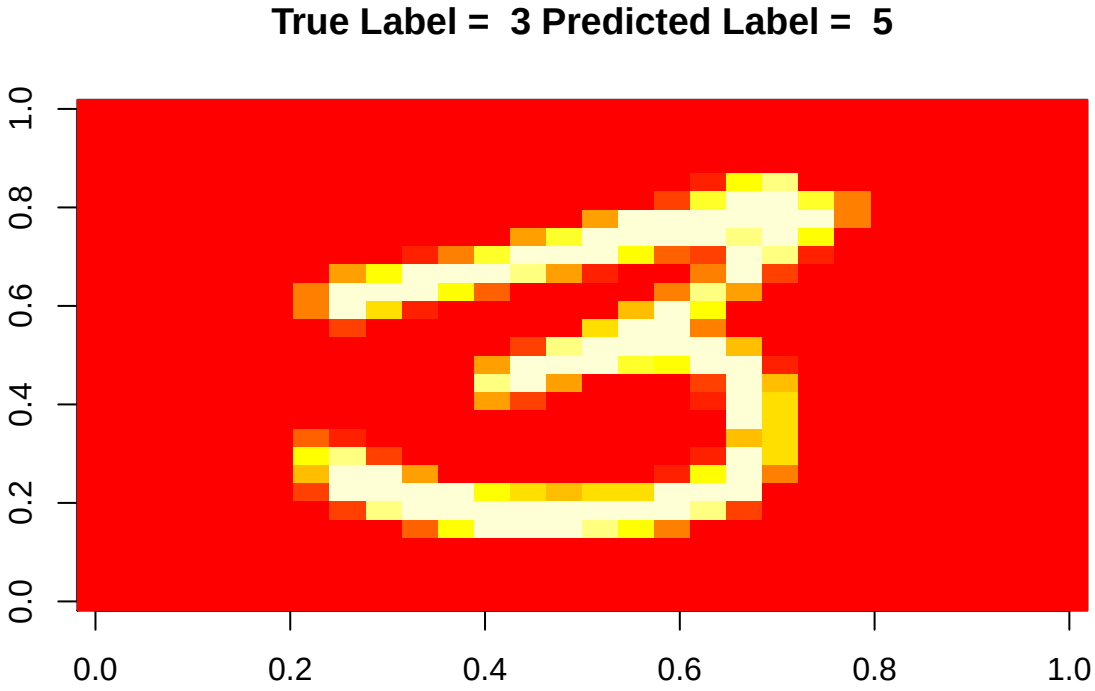

```
image(rotate(incorrect_0_1_b_matrix), main=paste('True Label = ', 0, 'Predicted Label = ', 1))
```




```
image(rotate(incorrect_3_5_a_matrix), main=paste('True Label = ', 5, 'Predicted Label = ', 3))
```




```
image(rotate(incorrect_3_5_b_matrix), main=paste('True Label = ', 3, 'Predicted Label = ', 5))
```




## Section 4: Modeling

```
print_confusion_matrix = function(labels, labels_pred){
  # Print the confusion matrix
  # Keyword args:
  #   labels: vector containing the true labels (y(i))
```

```
  #   labels_pred: vector containing predicted labels (\hat{y}^(i))
  tab = as.matrix(table(labels, labels_pred))
  print(tab)
}

accuracy = function(labels, labels_pred){
  # Compute the prediction accuracy
  # Keyword args:
  #   labels: vector containing the true labels (y(i))
  #   labels_pred: vector containing predicted labels (\hat{y}^(i))
  # Returns:
  #   acc: fraction of predicted labels that match true labels, as a number between 0 and 1
  tab = as.matrix(table(labels, labels_pred))
  true_negatives = tryCatch(tab[1, 1], error = function(e) 0)
  true_positives = tryCatch(tab[2, 2], error = function(e) 0)
  return((true_positives + true_negatives)/sum(tab))
}

model = function(train_data, train_labels, test_data, test_labels, alpha){
  # Train and evaluate a model.
  # Keyword args:
  #   train_data, train_labels: samples to be used for training, and for computing training accuracy
  #   test_data, test_samples: unseen samples for computing test accuracy
  #   alpha: learning rate hyperparameter for training
  # Returns:
  #   theta: learned model parameters
  #   train_acc: prediction accuracy on training data
  #   test_acc: prediction accuracy on test data
  #   train_pred: vector of predictions on training data
  #   test_pred: vector of predictions on test data

  theta = train(data=train_data, labels=train_labels, alpha=alpha)
  train_pred = predict(theta=theta, data=train_data)
  test_pred = predict(theta=theta, data=test_data)

  train_acc = accuracy(train_labels, train_pred)
  test_acc = accuracy(test_labels, test_pred)

  return(list('theta'=theta,
              'train_acc'=train_acc,
              'test_acc'=test_acc,
              'train_pred'=train_pred,
              'test_pred'=test_pred))
}
```

```
# Setting seed to 1 to ensure reproducible results
set.seed(1)

model_0_1 = model(train_data=train_0_1_w_bias,
                  train_labels=train_labels_0_1,
                  test_data=test_0_1_w_bias,
                  test_labels=test_labels_0_1,
                  alpha=alpha)
model_3_5 = model(train_data=train_3_5_w_bias,
```

```
                  train_labels=train_labels_3_5,
                  test_data=test_3_5_w_bias,
                  test_labels=test_labels_3_5,
                  alpha=alpha)

print('printing confusion matrices')
```

```
## [1] "printing confusion matrices"
```

```
print('Training over 0/1')
```

```
## [1] "Training over 0/1"
```

```
print_confusion_matrix(train_labels_0_1, model_0_1$train_pred)
```

```
##       labels_pred
## labels   -1    1
##     -1 5900   23
##     1    29 6713
```

```
print('Training over 3/5')
```

```
## [1] "Training over 3/5"
```

```
print_confusion_matrix(train_labels_3_5, model_3_5$train_pred)
```

```
##       labels_pred
## labels   -1    1
##     -1 5887  244
##     1   472 4949
```

```
print('Test over 0/1')
```

```
## [1] "Test over 0/1"
```

```
print_confusion_matrix(test_labels_0_1, model_0_1$test_pred)
```

```
##       labels_pred
## labels   -1    1
##     -1  977    3
##     1     2 1133
```

```
print('Test over 3/5')
```

```
## [1] "Test over 3/5"
```

```
print_confusion_matrix(test_labels_3_5, model_3_5$test_pred)
```

```
##       labels_pred
## labels  -1   1
##     -1 984  26
##     1   82 810
```

From the above, we can see that we trained two models one time, one on the 0/1 data set and the other on the 3/5 data set. With a fixed alpha of 0.01, the training accuracy of the 0/1 model is 0.9958942, and the test accuracy is 0.9976359.

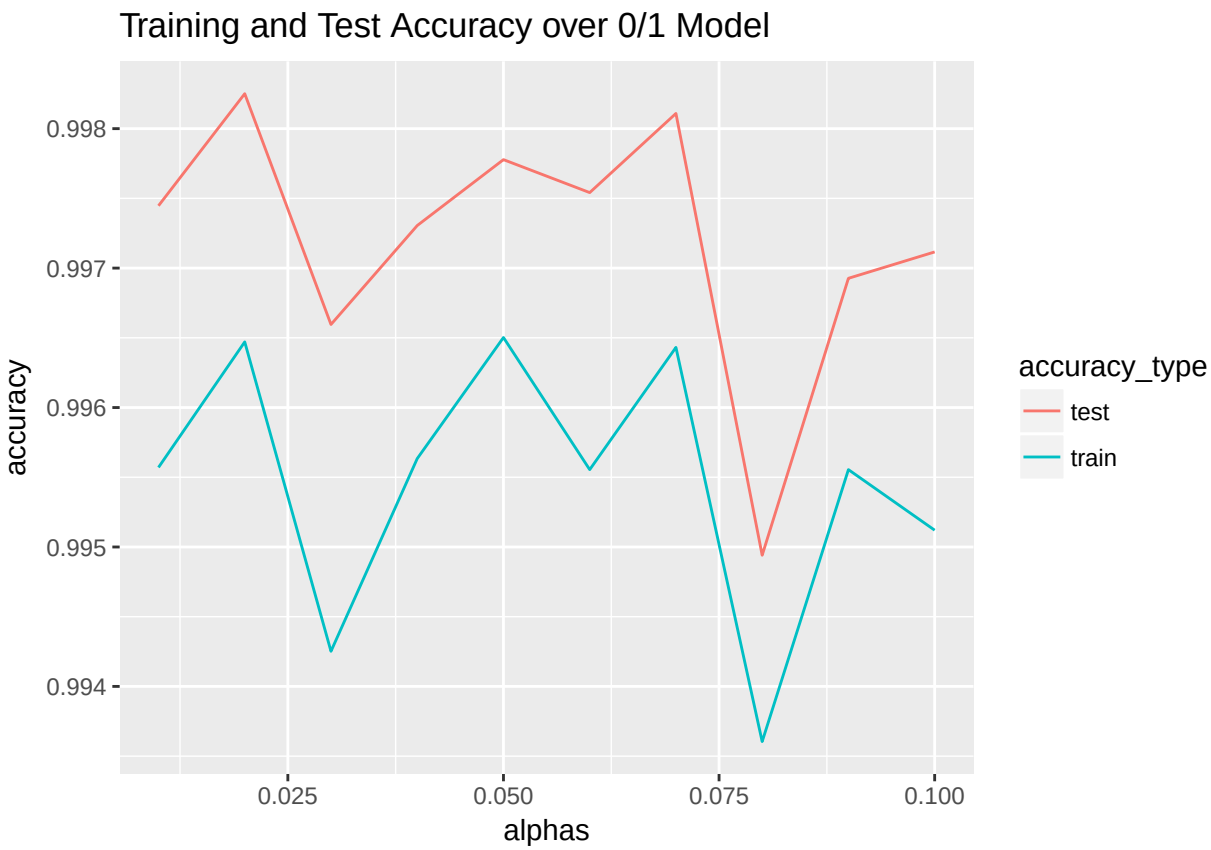
For the 3/5 data set, the training accuracy is 0.9380194, and the test accuracy is 0.9976359. Above we also print the confusion matrix so we can also see precision/recall.

### Evaluate models over different values of alpha

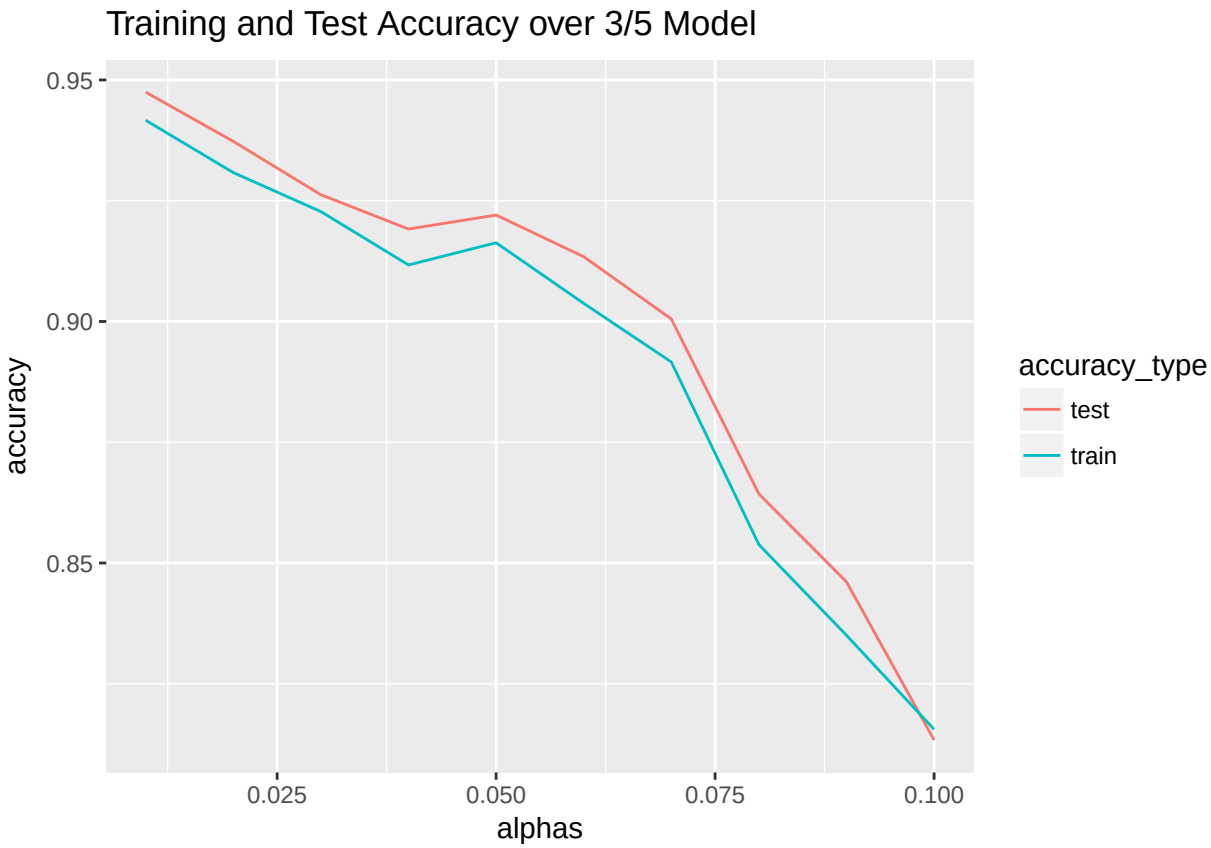
In the next section, we will evalute models over different values of $\alpha$, the learning parameter. Furthermore, since there can be substantial variance in the accuracy, we will evaluate the model 10 times and take the average accuracy over the 10 runs.

```
# Now vary alpha
alphas = c(0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09, 0.10)
num_runs = 10

train_accs_0_1 = c()
train_accs_3_5 = c()
test_accs_0_1 = c()
test_accs_3_5 = c()
for(alpha in alphas){
  train_acc_0_1 = 0
  train_acc_3_5 = 0
  test_acc_0_1 = 0
  test_acc_3_5 = 0
  for(i in 1:num_runs){
    model_0_1 = model(train_data=train_0_1_w_bias,
                      train_labels=train_labels_0_1,
                      test_data=test_0_1_w_bias,
                      test_labels=test_labels_0_1,
                      alpha=alpha)
    model_3_5 = model(train_data=train_3_5_w_bias,
                      train_labels=train_labels_3_5,
                      test_data=test_3_5_w_bias,
                      test_labels=test_labels_3_5,
                      alpha=alpha)
    train_acc_0_1 = train_acc_0_1 + model_0_1$train_acc
    train_acc_3_5 = train_acc_3_5 + model_3_5$train_acc
    test_acc_0_1 = test_acc_0_1 + model_0_1$test_acc
    test_acc_3_5 = test_acc_3_5 + model_3_5$test_acc
  }
  # Take the average
  train_acc_0_1 = train_acc_0_1 / num_runs
  train_acc_3_5 = train_acc_3_5 / num_runs
  test_acc_0_1 = test_acc_0_1 / num_runs
  test_acc_3_5 = test_acc_3_5 / num_runs

  train_accs_0_1 = c(train_accs_0_1, train_acc_0_1)
  train_accs_3_5 = c(train_accs_3_5, train_acc_3_5)
  test_accs_0_1 = c(test_accs_0_1, test_acc_0_1)
  test_accs_3_5 = c(test_accs_3_5, test_acc_3_5)
}

accuracy_type = c(rep('train', length(alphas)), rep('test', length(alphas)))
accuracy_0_1 = c(train_accs_0_1, test_accs_0_1)
accuracy_3_5 = c(train_accs_3_5, test_accs_3_5)

df_0_1 = data.frame(alphas=alphas, accuracy_type=accuracy_type, accuracy=accuracy_0_1)
df_3_5 = data.frame(alphas=alphas, accuracy_type=accuracy_type, accuracy=accuracy_3_5)
```

```
max_test_0_1 = max(test_accs_0_1)
alpha_max_test_0_1 = alphas[which.max(test_accs_0_1)]
max_test_3_5 = max(test_accs_3_5)
alpha_max_test_3_5 = alphas[which.max(test_accs_3_5)]

# Plotting scatter plot accuracy
ggplot(df_0_1, aes(x=alphas, y=accuracy)) +
  geom_line(aes(color=accuracy_type, group=accuracy_type)) +
  ggtitle('Training and Test Accuracy over 0/1 Model')
```




```
ggplot(df_3_5, aes(x=alphas, y=accuracy)) +
  geom_line(aes(color=accuracy_type, group=accuracy_type)) +
  ggtitle('Training and Test Accuracy over 3/5 Model')
```




**Did you observe any difference in the accuracies for 0/1 and 3/5 models? If so, explain why do you think the difference might be.**

Yes, I did observe differences in the accuracies for the 0/1 and 3/5 models. In general the performance of the 3/5 model is poorer than the performance of the 0/1 model, especially at higher ranges of alpha. This may be because distinguishing between 3's vs 5's is harder than distinguishing between 0's and 1's since 3's and 5's look more similar.

**Report the best test accuracy you could achieve for 0/1, and the best test accuracy for 3/5. These may be at different learning rate settings.**

The best test accuracy achieved for 0/1 model is 0.9982506 with a corresponding $\alpha$ of 0.02. The best test accuracy achieved for 3/5 model is 0.9474763 with a corresponding $\alpha$ of 0.01.

**This assignment deals with binary classification. Explain what you would do if you had more than two classes to classify using Logistic Regression (e.g. with a combined 0/1/3/5 dataset).**

One way to do this is to take a one-vs-all appraoch. More specifically in this case, we have 4 different classes: 0, 1, 3, 5. We would train 4 logistic regression classifiers, where in each classifier we would take one class and label as the "positive" class, and every other class is the "negative" class. In particular, in this case it would be:

1. 0 vs 1, 3, 5
2. 1 vs 0, 3, 5
3. 3 vs 0, 1, 5

4. 5 vs 0, 1, 3

Then, to make a prediction about any particular data point, you would take the predicted class as the class with the highest probability. In other words, out of the 4 models above, you would predict the class with the highest probability when that class was the "positive" class. To put this even more concretely, let's say for a particular data point, the predicted probabilities from the 4 logistic regression classifiers above were:

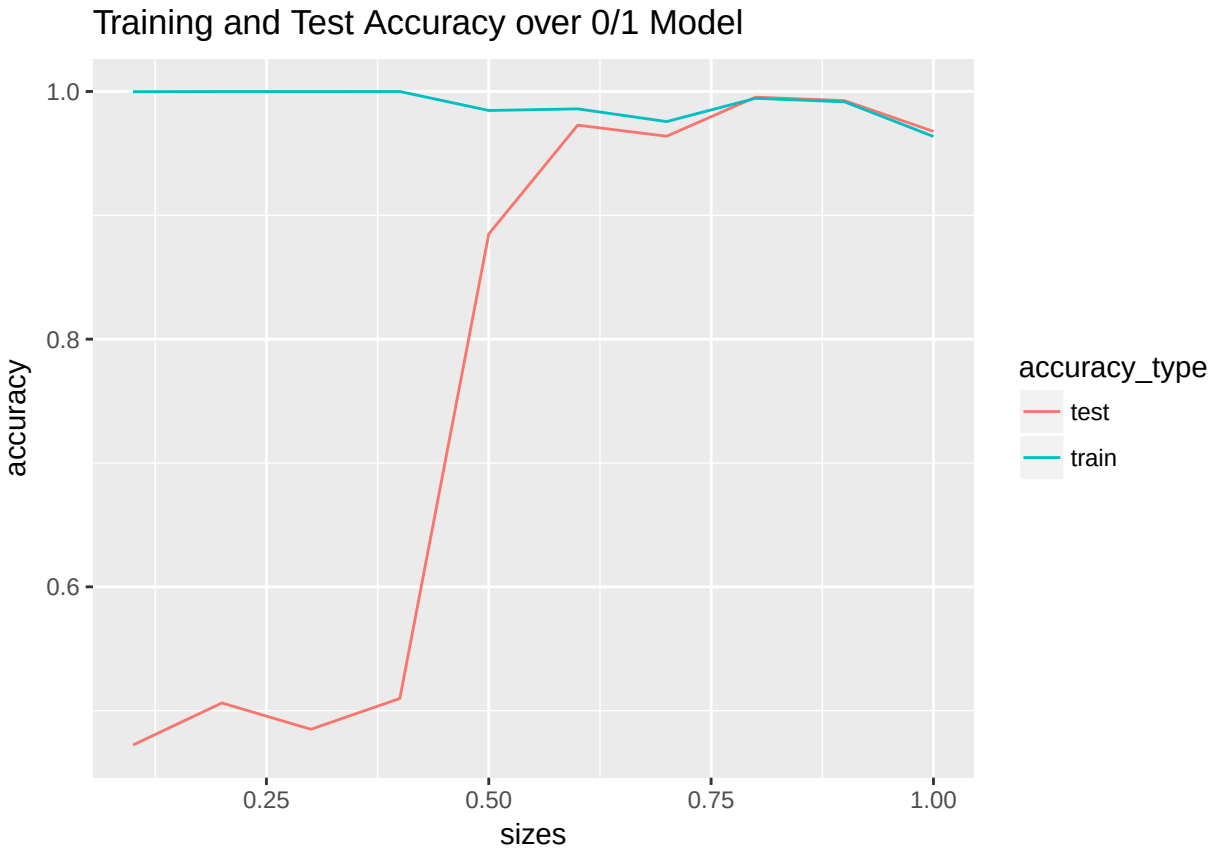
1. 0 vs 1, 3, 5 – 0.3
2. 1 vs 0, 3, 5 – 0.6
3. 3 vs 0, 1, 5 – 0.2
4. 5 vs 0, 1, 3 – 0.4

In this case, the logistic classifier where 1 was the "positive" class and the rest were the "negative" class performed the best, with a probability of the "positive" class being 0.6. Thus, we would classify this point as 1.

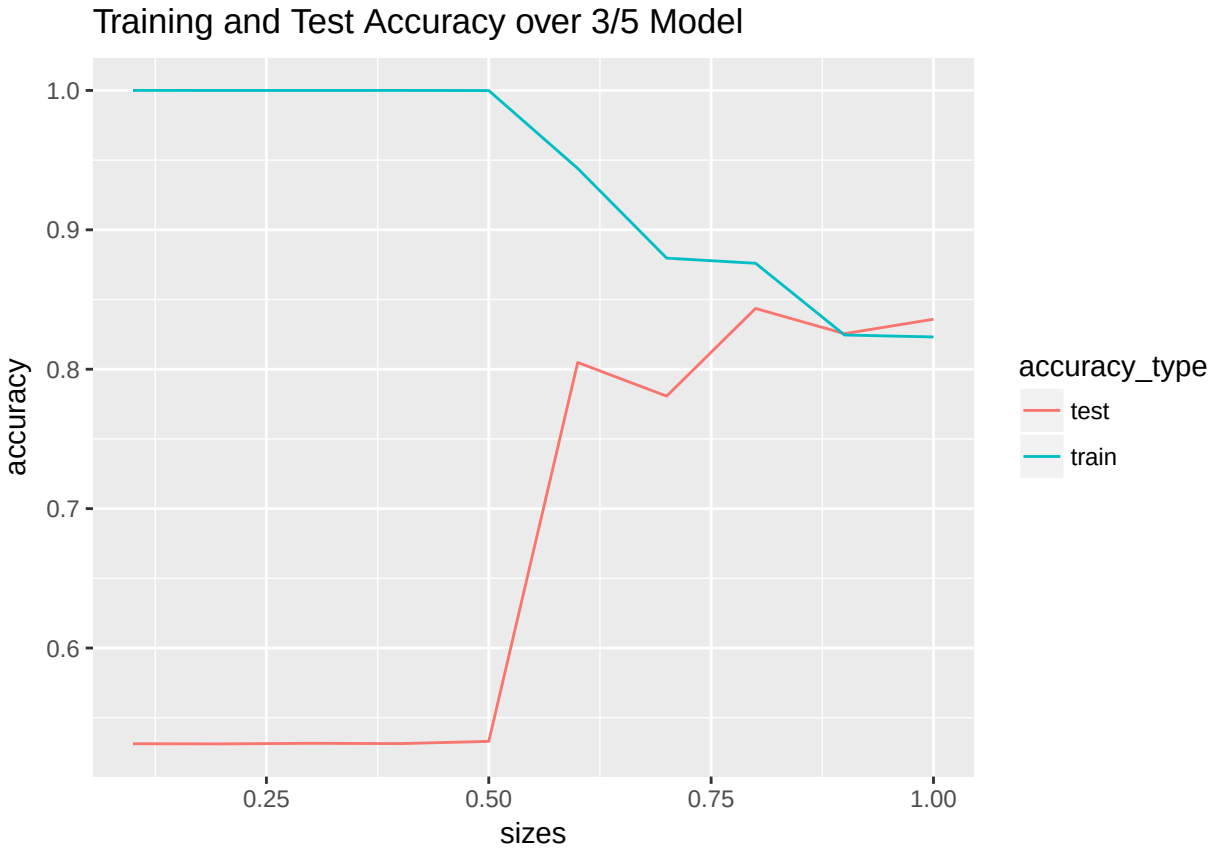
## Section 5: Learning Curves

```
sizes = c(0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0)
alpha = 0.1
num_runs = 10

train_accs_0_1 = c()
train_accs_3_5 = c()
test_accs_0_1 = c()
test_accs_3_5 = c()
for(size in sizes){
  train_acc_0_1 = 0
  train_acc_3_5 = 0
  test_acc_0_1 = 0
  test_acc_3_5 = 0
  for(i in 1:num_runs){
    data_idx_0_1 = sample(ncol(train_0_1_w_bias) * size)
    data_idx_3_5 = sample(ncol(train_3_5_w_bias) * size)
    model_0_1 = model(train_data=train_0_1_w_bias[, data_idx_0_1],
                      train_labels=train_labels_0_1[data_idx_0_1],
                      test_data=test_0_1_w_bias,
                      test_labels=test_labels_0_1,
                      alpha=alpha)
    model_3_5 = model(train_data=train_3_5_w_bias[, data_idx_3_5],
                      train_labels=train_labels_3_5[data_idx_3_5],
                      test_data=test_3_5_w_bias,
                      test_labels=test_labels_3_5,
                      alpha=alpha)
    train_acc_0_1 = train_acc_0_1 + model_0_1$train_acc
    train_acc_3_5 = train_acc_3_5 + model_3_5$train_acc
    test_acc_0_1 = test_acc_0_1 + model_0_1$test_acc
    test_acc_3_5 = test_acc_3_5 + model_3_5$test_acc
  }
  # Take the average
  train_acc_0_1 = train_acc_0_1 / num_runs
  train_acc_3_5 = train_acc_3_5 / num_runs
  test_acc_0_1 = test_acc_0_1 / num_runs
```

```
  test_acc_3_5 = test_acc_3_5 / num_runs

  train_accs_0_1 = c(train_accs_0_1, train_acc_0_1)
  train_accs_3_5 = c(train_accs_3_5, train_acc_3_5)
  test_accs_0_1 = c(test_accs_0_1, test_acc_0_1)
  test_accs_3_5 = c(test_accs_3_5, test_acc_3_5)
}

accuracy_type = c(rep('train', length(sizes)), rep('test', length(sizes)))
accuracy_0_1 = c(train_accs_0_1, test_accs_0_1)
accuracy_3_5 = c(train_accs_3_5, test_accs_3_5)

df_0_1 = data.frame(sizes=sizes, accuracy_type=accuracy_type, accuracy=accuracy_0_1)
df_3_5 = data.frame(sizes=sizes, accuracy_type=accuracy_type, accuracy=accuracy_3_5)

# Plotting scatter plot accuracy
ggplot(df_0_1, aes(x=sizes, y=accuracy)) +
  geom_line(aes(color=accuracy_type, group=accuracy_type)) +
  ggtitle('Training and Test Accuracy over 0/1 Model')
```




```
ggplot(df_3_5, aes(x=sizes, y=accuracy)) +
  geom_line(aes(color=accuracy_type, group=accuracy_type)) +
  ggtitle('Training and Test Accuracy over 3/5 Model')
```




**Observation on Learning Curves**

In both the 0/1 and 3/5 model at lower sample sizes, the training accuracy is much higher (close to 1) compared to the test set. This makes sense because at lower sample sizes, the model is severely overfitting on the training set and so the training accuracy is very high (close to 1), but the model does relatively poorly on the test set.

With higher sample sizes, the model is less overfitting to just a small sample of the data so we see the test accuracy increase (pretty stark difference at around size percentage of 50 percent). Furthermore, notice that the training accuracy does decrease with size, which makes sense as well. At lower sample sizes, the model is, as said previously, very much overfitting to a small sample size. So for those small sample sizes, the model does very well on the training set. However, at larger sample sizes, the model can't overfit as much on small sample sizes, so the training accuracy goes down.

Again, we observe that in general the 3/5 model performance generally does worse than the 0/1 model; probably again because 3's and 5's are much harder to distinguish from each other than 0's vs 1's.