Homework 1

CS 498, Spring 2018, Xiaoming Ji

Problem 1

Part A

Build a simple naive Bayes classifier to classify this data set. We will use 20% of the data for evaluation and the other 80% for training. There are a total of 768 data-points.

We use a normal distribution to model each of the class-conditional distributions.

```
#Prepare training and testing data
library(readr)
all_data  = read.csv("pima-indians-diabetes.data.csv", header = FALSE)
```

We define a function to train naive Bayes model with Gaussian model and Bernoulli model (for later use) by calculating p(y=class), mean and standard deviation of feature variables for each class.

```
#Gaussian model
gnb.fit = function(features, labels) {
  model      = list("p_y", "mu","sigma")
  class_val  = unique(labels)
  class_num  = max(class_val) + 1
  record_num = length(labels)

  p_y       = rep(0, class_num)
  mu        = matrix(0, class_num, dim(features)[2])
  sigma     = matrix(0, class_num, dim(features)[2])

  for (y in class_val) {
    indexes = labels == y
    p_y[y + 1]  = length(labels[indexes]) / record_num

    x             = features[indexes,]
    mu[y + 1,]    = sapply(x, mean, na.rm=TRUE)
    sigma[y + 1,] = sapply(x, sd, na.rm=TRUE)
  }

  model$p_y = p_y
  model$mu = mu
  model$sigma = sigma

  return (model)
}

#Bernoulli model
bnb.fit = function(features, labels) {
  model = list("p_y", "p_x")
  class_val  = unique(labels)
  class_num  = max(class_val) + 1
  record_num = length(labels)
```

```
  p_y = rep(0, class_num)
  p_x = matrix(0, class_num, dim(features)[2])

  for (y in class_val){
    indexes = labels == y
    p_y[y + 1]  = length(labels[indexes]) / record_num

    x           = features[indexes,]
    p_x[y + 1,] = colSums(x > 0) / length(indexes)
  }

  model$p_y = p_y
  model$p_x = p_x

  return (model)
}
```

Since we use normal distribution for each feature x_j, the probability of x_j given class y_k is given as,

$$p(x_j|y_k) = \frac{1}{\sigma_k\sqrt{2\pi}} e^{-\frac{(x-\mu_k)^2}{2\sigma_k^2}}$$

where μ_k and σ_k are mean and standard deviation of feature x_j for a given class y_k. They are calculated by function *gnb.fit*.

To calculate the log probability of one class y_k, we use formula,

$$\sum_j log(x_j|y_k) + logp(y_k)$$

$$= -\sum_j (log\sigma_k + log\sqrt{2\pi} + \frac{(x-\mu_k)^2}{2\sigma_k^2}) + logp(y_k)$$

Given $log\sqrt{2\pi}$ is constant, we have,

$$\propto -\sum_j (log\sigma_k + \frac{(x-\mu_k)^2}{2\sigma_k^2}) + logp(y_k)$$

We define functions to predict the class given a model and features.

```
gnb.predict = function(model, x) {
  class_num = length(model$p_y)
  y_score = matrix(0, dim(x)[1],class_num)

  for (i in 1:class_num) {
    scale       = (t(x)-model$mu[i,])/model$sigma[i,]
    logs        = -(log(model$sigma[i,]) + (1/2) * scale ^ 2)
    y_score[,i] = colSums(logs, na.rm = TRUE) + log(model$p_y[i])
  }

  return (max.col(y_score) - 1)
}

bnb.predict = function(model, x) {
```

```
    class_num = length(model$p_y)
    y_score = matrix(0, dim(x)[1], class_num)
    for (i in 1:class_num){
      logs = t(x) * log(model$p_x[i,]) + t(1 - x) * log(1 - model$p_x[i,])
      y_score[,i] =  colSums(logs, na.rm = TRUE)+ log(model$p_y[i])
    }

    return (max.col(y_score) - 1)
}
```

Let's train and check the accuracy of the classifier on the 20% evaluation data. We will do 10-fold cross-validation and select the best model for testing evaluation data.

```
#Cross validate and return best model
cross_validation = function(x, y, fold_num, model = "Gaussian") {
  func_map = list("Gaussian" = list(fit = gnb.fit, predict = gnb.predict),
                  "Bernoulli" = list(fit = bnb.fit, predict = bnb.predict))

  train_num = length(y)
  accuracy  = rep(0, fold_num)
  models    = list()


  valid_num = round(train_num / fold_num)
  for (i in 1:fold_num){
    index_from = (i - 1) * valid_num + 1
    index_to   = i * valid_num + 1
    if(index_to > train_num) index_to = train_num

    valid_indexes = index_from : index_to
    train_x = x[-valid_indexes, ]
    train_y = y[-valid_indexes]
    valid_x = x[valid_indexes, ]
    valid_y = y[valid_indexes]

    models[[i]] = func_map[[model]]$fit(train_x, train_y)
    pred_y = func_map[[model]]$predict(models[[i]], valid_x)

    accuracy[i] = sum(pred_y == valid_y) / length(valid_y)
    #print(accuracy[i])
  }

  #print(which.max(accuracy))
  return (models[[which.max(accuracy)]])
}

set.seed(19720816)
fold_num = 10
train_indexes = sample(1:dim(all_data)[1], round(dim(all_data)[1] * 0.8))

best_model = cross_validation(all_data[train_indexes,-c(9)], all_data[train_indexes,c(9)], 10)

test_x = all_data[-train_indexes,-c(9)]
test_y = all_data[-train_indexes,c(9)]
```

```
pred_y = gnb.predict(best_model, test_x)
best_accuracy = sum(pred_y == test_y)/length(test_y)
```

We got accuracy: 0.8116883

Part B

We will adjust our code so that, for attribute 3 (Diastolic blood pressure), attribute 4 (Triceps skin fold thickness), attribute 6 (Body mass index), and attribute 8 (Age), it regards a value of 0 as a missing value when estimating the class-conditional distributions, and the posterior. R uses a special number NA to flag a missing value.

```
all_x_na = all_data

for (i in c(3, 5, 6, 8))
{
  index = all_x_na[, i]==0
  all_x_na[index, i] = NA
}

set.seed(19720816)
fold_num = 10
train_indexes = sample(1:dim(all_x_na)[1], round(dim(all_x_na)[1] * 0.8))

best_model = cross_validation(all_x_na[train_indexes,-c(9)], all_x_na[train_indexes,c(9)], 10)

test_x = all_x_na[-train_indexes,-c(9)]
test_y = all_x_na[-train_indexes,c(9)]
pred_y = gnb.predict(best_model, test_x)
best_accuracy = sum(pred_y == test_y)/length(test_y)
```

We got accuracy: 0.8051948. The result don't have much difference compared with the previous one.

Part C

Use the caret and klaR packages to build a naive Bayes classifier for this data, assuming that no attribute has a missing value. We will do 10-fold cross-validation and test the accuracy of the classifier on the held out 20%.

```
library(caret)

train_indexes = createDataPartition(y = all_data[, 9], p = 0.8, list = FALSE)
train_x = all_data[train_indexes,-c(9)]
train_y = as.factor(all_data[train_indexes,c(9)])
model   = train(train_x, train_y, 'nb', trControl = trainControl(method='cv', number=10))

test_x = all_data[-train_indexes,-c(9)]
test_y = all_data[-train_indexes,c(9)]
pred_y = predict(model, newdata = test_x)
accuracy = sum(pred_y == test_y)/length(test_y)
```

We got accuracy: 0.7189542

Part D

Now we use SVMLight to train and evaluate an SVM to classify this data.

```
library(klaR)
```

```
## Warning: package 'MASS' was built under R version 3.4.3
```

```
train_indexes = createDataPartition(y = all_data[, 9], p = 0.8, list = FALSE)
train_x = all_data[train_indexes,-c(9)]
train_y = all_data[train_indexes,c(9)]

model = svmlight(train_x, train_y, pathsvm="./svmlight")

test_x = all_data[-train_indexes,-c(9)]
test_y = all_data[-train_indexes,c(9)]
pred_y = predict(model, newdata = test_x)
accuracy = sum(pred_y$class == test_y)/length(test_y)
```

We got accuracy: 0.7581699

Problem 2

We make a function to construct a bounding box so that the horizontal (resp. vertical) range of ink pixels runs the full horizontal (resp. vertical) range of the box.

```
#Stretch image to new size
stretch_image = function(img, new_width, new_height) {
  new_img = apply(img, 2, function(y){return (spline(y, n = new_height)$y)})
  new_img = t(apply(new_img, 1, function(y){return (spline(y, n = new_width)$y)}))

  new_img[new_img < 0] = 0
  new_img[new_img > 255] = 255
  new_img = round(new_img)

  return (new_img)
}

#Make stretched bounding box
sb_box = function(img, box_width, box_height) {
  col = colSums(img)
  row = rowSums(img)

  #Calculate image boundary and extract the bounding image
  top = bottom = left = right  = 0

  for (i in 1:length(row)) {if (row[i] != 0) { top = i; break}}
  for (i in length(row):1) {if (row[i] != 0) { bottom = i; break}}
  for (i in 1:length(col)) {if (col[i] != 0) { left = i; break}}
  for (i in length(col):1) {if (col[i] != 0) { right = i; break}}

  if(top >= bottom || left >= right){stop("bad image")}

  box_img = img[top:bottom, left:right]
```

```
  top    = (dim(img)[1] - box_height) / 2 + 1
  bottom = top + box_height - 1
  left   = (dim(img)[2] - box_width) / 2 + 1
  right  = left + box_width - 1

  ss_img = array(0, dim(img))
  ss_img[top:bottom, left:right] = (stretch_image(box_img, box_width, box_height))

  #Draw box
  ss_img[top, left:right]    = 255
  ss_img[bottom, left: right] = 255
  ss_img[top: bottom, left]    = 255
  ss_img[top: bottom, right]   = 255

  return (ss_img)
}
```

To improve performance for repetitive experiment, we process the original images and save the stretched bounding box images to files.

```
process_save = function(input_gzfile_name, output_file_name) {
  input_gzfile = file(input_gzfile_name, "rb")
  output_file = file(output_file_name, "wb")

  header = readBin(input_gzfile, integer(), n = 4,  endian = "big")
  writeBin(header, output_file, endian = "big")

  sb_images = list()
  original_images = list()

  for (i in 1:header[2]) {
    data = readBin(input_gzfile, integer(), size = 1, n = 28*28, endian = "big",
                   signed = FALSE)
    if(length(data) != 28*28) break
    original_images[[i]] = matrix(data, 28, 28);

    sb_images[[i]] = sb_box(original_images[[i]], 20, 20)
    writeBin(as.integer(sb_images[[i]]), output_file, size = 1)
  }

  close(input_gzfile)
  close(output_file)
}

process_save("./MNIST/train-images-idx3-ubyte", "./MNIST/train-images-idx3-ubyte-sbb")
process_save("./MNIST/t10k-images-idx3-ubyte", "./MNIST/t10k-images-idx3-ubyte-sbb")
```

Load data from saved file.

```
load_image_data = function(file_name){
  img_file = file(file_name, "rb")
  header = readBin(img_file, integer(), n = 4,  endian = "big")
```

```
  data = matrix(0, header[2], 28 * 28)
  for (i in 1:header[2]) {
    img = readBin(img_file, integer(), size = 1, n = 28*28, endian = "big",
                  signed = FALSE)
    if(length(img) != 28*28) break
    data[i,] = img
  }
  close(img_file)

  return (data)
}

load_label_data = function(file_name){
  label_file = file(file_name, "rb")
  header = readBin(label_file, integer(), n = 2,  endian = "big")

  data = readBin(label_file, integer(), size = 1, n = header[2], endian = "big",
                  signed = FALSE)
  close(label_file)

  return (data)
}

  train_data = load_image_data("./MNIST/train-images-idx3-ubyte")
  test_data = load_image_data("./MNIST/t10k-images-idx3-ubyte")

  sbb_train_data = load_image_data("./MNIST/train-images-idx3-ubyte-sbb")
  sbb_test_data = load_image_data("./MNIST/t10k-images-idx3-ubyte-sbb")

  train_label = load_label_data("./MNIST/train-labels-idx1-ubyte")
  test_label = load_label_data("./MNIST/t10k-labels-idx1-ubyte")

  #Threshhold image to make it binary value (ink and paper)
  bin_train_data = train_data
  bin_test_data = test_data
  bin_train_data[bin_train_data <= 128] = 0
  bin_train_data[bin_train_data > 128] = 1
  bin_test_data[bin_test_data <= 128] = 0
  bin_test_data[bin_test_data > 128] = 1

  sbb_bin_train_data = sbb_train_data
  sbb_bin_test_data  = sbb_test_data
  sbb_bin_train_data[sbb_bin_train_data <= 128] = 0
  sbb_bin_train_data[sbb_bin_train_data > 128] = 1
  sbb_bin_test_data[sbb_bin_test_data <= 128] = 0
  sbb_bin_test_data[sbb_bin_test_data > 128] = 1
```

Part A

Investigate classifying MNIST using naive Bayes. We use 20 x 20 for your bounding box dimensions.

```
train_df = as.data.frame(train_data)
bin_train_df = as.data.frame(bin_train_data)
```

```
sbb_train_df = as.data.frame(sbb_train_data)
sbb_bin_train_df = as.data.frame(sbb_bin_train_data)

best_model = cross_validation(train_df, train_label, 10)
pred_y = gnb.predict(best_model, test_data)
gaussian_accuracy = sum(pred_y == test_label)/length(test_label)

best_model = cross_validation(bin_train_df, train_label, 10)
pred_y = gnb.predict(best_model, bin_test_data)
bin_gaussian_accuracy = sum(pred_y == test_label)/length(test_label)

best_model = cross_validation(sbb_train_df, train_label, 10)
pred_y = gnb.predict(best_model, sbb_test_data)
gaussian_sbb_accuracy = sum(pred_y == test_label)/length(test_label)

best_model = cross_validation(sbb_bin_train_df, train_label, 10)
pred_y = gnb.predict(best_model, sbb_bin_test_data)
bin_gaussian_sbb_accuracy = sum(pred_y == test_label)/length(test_label)

best_model = cross_validation(bin_train_df, train_label, 10, model = "Bernoulli")
pred_y = bnb.predict(best_model, bin_test_data)
bernoulli_accuracy = sum(pred_y == test_label)/length(test_label)

best_model = cross_validation(sbb_bin_train_df, train_label, 10, model = "Bernoulli")
pred_y = bnb.predict(best_model, sbb_bin_test_data)
bernoulli_sbb_accuracy = sum(pred_y == test_label)/length(test_label)
```

The accuracy values for the six combinations of Gaussian (original and thresholding) v. Bernoulli distributions and untouched images v. stretched bounding boxes.

Accuracy	Gaussian	Gaussian (Thresholding)	Bernoulli
untouched images	0.6771	0.726	0.7448
stretched bounding box	0.8266	0.821	0.4578

- From the results, we can see Bernoulli distribution is better for untouched pixels and Gaussian distribution is better for stretched bounding box images. The reason is Gaussian distribution model fit the stretched bounding box (image) better than untouched image. And Bernoulli distribution model fit the untouched image better than stretched bounding box (image).
- We also notice that Gaussian distribution used against (unthresholded) stretched bounding box gives us the best result 0.8266.

Part B

Investigate classifying MNIST using a decision forest. For this we will use *h2o* library. For forest construction, will try out and compare the combinations of parameters shown in the table (i.e. depth of tree, number of trees, etc.) by listing the accuracy for each of the following cases: untouched raw pixels; stretched bounding box.

```
library(h2o)

calculate_accuracy = function(train_frame, test_frame, ntrees, max_depths) {
  accuracies = matrix(0, length(ntrees),  length(max_depths))
```

```
  for (ntree in 1:length(ntrees)) {
    for (depth in 1:length(max_depths)) {
      rf = h2o.randomForest(y = "label", training_frame = train_frame, ntrees = ntrees[ntree],
                            max_depth = max_depths[depth])
      prediction = h2o.predict(rf, newdata = test_frame[-dim(test_frame)[2]])

      accuracies[ntree, depth] = sum(prediction[,1] == test_frame["label"]) / dim(test_frame)[1]
    }
  }

  return (accuracies)
}

h2o.init(nthreads=-1, max_mem_size = "2G")
h2o.removeAll()

h2o_train_data = as.h2o(data.frame(train_data, label=as.factor(train_label)))
h2o_test_data = as.h2o(data.frame(test_data, label=as.factor(test_label)))
h2o_sbb_train_data = as.h2o(data.frame(sbb_train_data, label=as.factor(train_label)))
h2o_sbb_test_data = as.h2o(data.frame(sbb_test_data, label=as.factor(test_label)))

accuracies = calculate_accuracy(h2o_train_data, h2o_test_data, c(10, 20, 30), c(4, 8, 16))
sbb_accuracies = calculate_accuracy(h2o_sbb_train_data, h2o_sbb_test_data, c(10, 20, 30), c(4, 8, 16))

h2o.shutdown(prompt=FALSE)
```

Untouched raw pixels	depth = 4	depth = 8	depth = 16
#trees = 10	0.8583	0.9292	0.9552
#trees = 20	0.8668	0.9378	0.9616
#trees = 30	0.8747	0.9402	0.965

Stretched bounding box	depth = 4	depth = 8	depth = 16
#trees = 10	0.858	0.9454	0.9593
#trees = 20	0.8653	0.949	0.9672
#trees = 30	0.8646	0.9508	0.9688

We notice that,

- more and deeper trees give us better results.
- untouched and stretched bounding box have similar results.