CS498 Applied Machine Learning HW1 Name: Yu Che Wang / Netid: yuchecw2

Three accuracies for 1A, 1B, 1D

1A

accuracy_vec = [0.6493506 0.7337662 0.6753247 0.6688312 0.6883117 0.6883117 0.7272727 0.6948052 0.7142857 0.6428571]

⇒accuracy = 0.6883117

1B

accuracy_vec = [0.6428571 0.6883117 0.6688312 0.6753247 0.6818182 0.6493506 0.6103896 0.7272727 0.6493506 0.6038961]

⇒accuracy = 0.6597403

1D

accuracy_vec = [0.6078431 0.7189542 0.6862745 0.6732026 0.6928105 0.6601307 0.6928105 0.6797386 0.6862745 0.6078431]

⇒accuracy = 0.6705882

Code

Evaluations

```
# Part 1B
# Read CSV into R
data = read.csv(file="Desktop/pima-indians-diabetes.csv")
accuracy_vec = seq(1, 10, by=1)
train_ratio = 0.8
attributes = c(3, 4, 6, 8) #blood pressure, skin thickness, BMI, age
param = matrix(, nrow=8, ncol=4)

for (iter in seq(1, 10, by=1)) {
  # Test-train spilts
  train_test_list = test_train_split(data, train_ratio, iter)
  train_data = train_test_list[[1]]
  test_data = train_test_list[[2]]
  # Calculate prior probabilities
  outcome = train_data[,9]
  n_positive = length(outcome[outcome==1])
  n_negative = length(outcome[outcome==0])
  P_positive = n_positive/length(outcome)
  P_negative = n_negative/length(outcome)
  # Calculate likelihood
  # Calculate normal distribution parameter for each feature
  for (attribute in attributes) {
    feature = train_data[,attribute]
    # Ignore zero entries aka missing values
    # I don't change 0 entries to NA, ignore those entries in this step
    remain_outcome = outcome[feature!=0]
    feature = feature[feature!=0]
    param[attribute, 1] = mean(feature[remain_outcome==1])   # positive_feature_mean
    param[attribute, 2] = var(feature[remain_outcome==1])    # positive_feature_var
    param[attribute, 3] = mean(feature[remain_outcome==0])   # negative_feature_mean
    param[attribute, 4] = var(feature[remain_outcome==0])    # negative_feature_var
  }
  confusion_matrix = estimate(test_data, param, attribute, P_positive, P_negative)
  n_correct = confusion_matrix[1,1] + confusion_matrix[2,2]
  accuracy = n_correct / nrow(test_data)
  accuracy_vec[iter] = accuracy
}
print(accuracy_vec)
print(mean(accuracy_vec))
```

Probability calculations

```
# Evalute log probability
evaluate_log_p = function(feature, feature_mean, feature_var) {
  log_p = - ((feature - feature_mean)^2)/(2*feature_var)
  log_p = log_p + log(1/sqrt(2*pi*feature_var))
  return(log_p)
}
```

Calculating the confusion matrix and test-train split

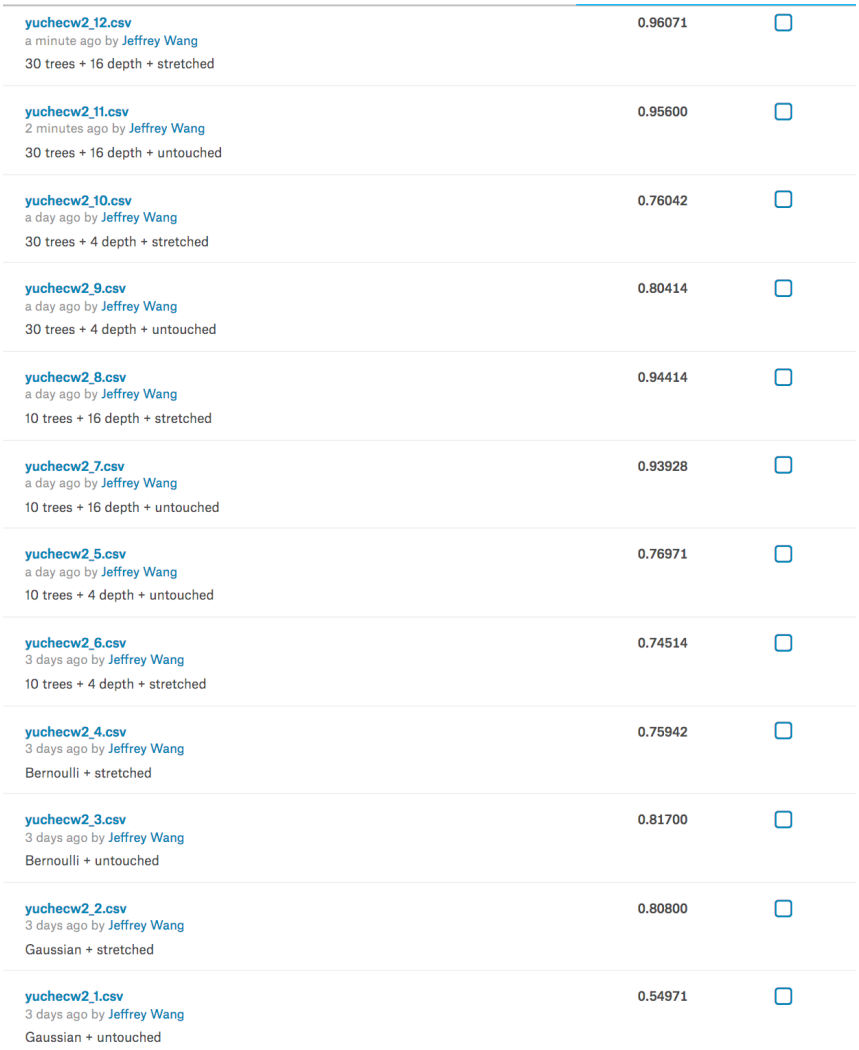
```
# Evaluate on test data and return the confusion matrix
estimate = function(test_data, param, attribute, P_positive, P_negative) {
  true_positive = 0
  false_positive = 0
  true_negative = 0
  false_negative = 0
  logP_positive = log(P_positive)
  logP_negative = log(P_negative)
  for(index in seq(1, nrow(test_data), by=1)) {
    positive_log_p = 0
    negative_log_p = 0
    # Calculate log likelihood
    for (attribute in attributes) {
      if (test_data[index, attribute] == 0) {
        # Ignore this attribute
        next
      } else {
        positive_log_p = positive_log_p + evaluate_log_p(test_data[index,attribute], param[attribute, 1], param[attribute, 2])
        negative_log_p = negative_log_p + evaluate_log_p(test_data[index,attribute], param[attribute, 3], param[attribute, 4])
      }
    }
    if (logP_positive + positive_log_p > logP_negative + negative_log_p) {
      if (test_data[index, 9] == 1) {
        true_positive = true_positive + 1
      } else {
        false_positive = false_positive + 1
      }
    } else {
      if (test_data[index, 9] == 0) {
        true_negative = true_negative + 1
      } else {
        false_negative = false_negative + 1
      }
    }
  }
  confusion_matrix = matrix(
    c(true_positive, false_positive, false_negative, true_negative),
    nrow = 2,
    ncol = 2,
    byrow = TRUE
  )
  return(confusion_matrix)
}

# Split the data
test_train_split = function(data, train_ratio, seed) {
  set.seed(seed)
  sample = sample(seq_len(nrow(data)), size=floor(train_ratio*nrow(data)))
  train_data = data[sample,]
  test_data = data[-sample,]
  return(list(train_data, test_data))
}
```

Table of accuracies for all 12 cases

Accuracies are obtained from validation

	Model	Accuracy
1	Gaussian + untouched	0.558107
2	Gaussian + stretched	0.811631
3	Bernoulli + untouched	0.833274
4	Bernoulli + stretched	0.810357
5	10 trees + 4 depth + untouched	0.753107
6	10 trees + 4 depth + stretched	0.751345
7	10 trees + 16 depth + untouched	0.937464
8	10 trees + 16 depth + stretched	0.942143
9	30 trees + 4 depth + untouched	0.797893
10	30 trees + 4 depth + stretched	0.784762
11	30 trees + 16 depth + untouched	0.955452
12	30 trees + 16 depth + stretched	0.958488



Random forest with 30 trees and 16 depth has the best performance since it detects more features.

40 mean images

1：Gaussian + untouched accuracy: 0.558107

2：Gaussian + stretched accuracy: 0.811631

3：Bernoulli + untouched accuracy: 0.833274

4：Bernoulli + stretched accuracy: 0.810357

Code

Main function

```python
import numpy as np
import pandas as pd
from sklearn.naive_bayes import BernoulliNB
from sklearn.naive_bayes import GaussianNB
from sklearn.ensemble import RandomForestClassifier
import matplotlib.pyplot as plt
import random
import math
import argparse
from helper import *

parser = argparse.ArgumentParser()
parser.add_argument('--model_name', default='Gaussian', type=str)
parser.add_argument('--preprocess', default=False, type=bool)
parser.add_argument('--num_epochs', default=10, type=int)
parser.add_argument('--test', default=True, type=bool)
parser.add_argument('--path', default='yuchecw2_1.csv', type=str)
parser.add_argument('--n_estimators', default=30, type=int)
parser.add_argument('--max_depth', default=16, type=int)
args = parser.parse_args()

train_data = pd.read_csv("MNIST/train.csv")
test_data = pd.read_csv("MNIST/test.csv")
train_data = pd.DataFrame.as_matrix(train_data)
test_data = pd.DataFrame.as_matrix(test_data)
if args.preprocess:
    train_data = preprocess(train_data, False)
if not args.test:
    ratio = 0.2
    num_data = train_data.shape[0]
    num_val = (int)(num_data*ratio)
    accuracy_list = []

    for epoch in range(args.num_epochs):
        random.seed(epoch)
        val_samples = random.sample(range(num_data), num_val)
        mask_array = np.ones((num_data, 1), dtype=bool)
        for i in val_samples:
            mask_array[i] = False

        label = np.empty((num_data-num_val, 1))
        feature = np.empty((num_data-num_val, train_data.shape[1]-1))
        idx = 0
        for i in range(num_data):
            if (mask_array[i]):
                label[idx] = train_data[i,0]
                feature[idx, :] = train_data[i,1:]
                idx = idx + 1

        val_label = train_data[val_samples, 0]
        val_feature = train_data[val_samples, 1:]

        # Create a model and train it
        if args.model_name == 'Gaussian':
            model = GaussianNB()
        elif args.model_name == 'Bernoulli':
            # Thresholding
            feature[feature>127] = 255
            feature[feature<=127] = 0
            val_feature[val_feature>127] = 255
            val_feature[val_feature<=127] = 0
            model = BernoulliNB()
        elif args.model_name == 'Random_Forest':
            model = RandomForestClassifier(n_estimators=args.n_estimators, max_depth=args.max_depth)
            val_feature[val_feature>127] = 255
            val_feature[val_feature<=127] = 0
            model = BernoulliNB()
        elif args.model_name == 'Random_Forest':
            model = RandomForestClassifier(n_estimators=args.n_estimators, max_depth=args.max_depth)

        label = np.ravel(label)
        fit_model = model.fit(feature, label)
        label_prediction = fit_model.predict(val_feature)
        num_correct = (val_label == label_prediction).sum()
        accuracy = num_correct/num_val
        accuracy_list.append(accuracy)
    print('Training and cross validation: uses %s model, preprocessed: %d, accuracy: %f'
                        %(args.model_name, args.preprocess, sum(accuracy_list)/len(accuracy_list)))
else:
    label = train_data[:,0]
    feature = train_data[:,1:]
    # Create a model and train it
    if args.model_name == 'Gaussian':
        model = GaussianNB()
    elif args.model_name == 'Bernoulli':
        feature[feature>127] = 255
        feature[feature<=127] = 0
        model = BernoulliNB()
    elif args.model_name == 'Random_Forest':
        model = RandomForestClassifier(n_estimators=args.n_estimators, max_depth=args.max_depth)

    label = np.ravel(label)
    fit_model = model.fit(feature, label)

    if args.preprocess:
        test_data = preprocess(test_data, True)
    test_prediction = fit_model.predict(test_data)
    df = pd.DataFrame(test_prediction)
    df_test_data = pd.DataFrame(test_data)
    df.to_csv(args.path)

    # Calculate class means
    if args.preprocess:
        original_size = 20 # box_size
    else:
        original_size = 28

    num_classes = 10
    mean = [None]*num_classes
    for label in range(num_classes):
        mean[label] = df_test_data[df[0]==label].mean()

    mean_matrix = []
    for label in range(num_classes):
        a = np.empty((original_size, original_size))
        for j in range(original_size):
            for i in range(original_size):
                if (mean[label][original_size*j+i] < 127):
                    a[j, i] = 255
                else:
                    a[j, i] = 0
        mean_matrix.append(a)

    plt.figure()
    for label in range(10):
        plt.subplot(1, 10, label+1)
        plt.imshow(mean_matrix[label], cmap='Greys')
        plt.axis('off')
    plt.show()
```

Helper functions

```python
import numpy as np
import pandas as pd
from sklearn.naive_bayes import BernoulliNB
from sklearn.naive_bayes import GaussianNB
import random
import math

def preprocess(data, test):
    box_size = 20
    original_size = 28
    if not test:
        streched_data = np.empty((data.shape[0], box_size*box_size+1))
    else:
        # No label column
        streched_data = np.empty((data.shape[0], box_size*box_size))
    for k in range(data.shape[0]):
        # Label
        if not test:
            streched_data[k, 0] = data[k, 0]
        # Find the margin
        bottom = find_bottom(data, k, original_size)
        top = find_top(data, k, original_size)
        num_rows = bottom-top+1
        right = find_right(data, k, original_size, test)
        left = find_left(data, k, original_size, test)
        num_cols = right-left+1
        # Find the bounding box
        bounding_box = np.empty((num_rows, num_cols))
        for j in range(top, bottom+1):
            for i in range(left, right+1):
                if not test:
                    bounding_box[j-top, i-left] = data[k, original_size*j+i+1]
                else:
                    bounding_box[j-top, i-left] = data[k, original_size*j+i]
        # plt.subplot(1,2,1)
        # plt.matshow(bounding_box)
        # plt.title("#%d bounding box"%(k))
        # Resize the bounding box
        streched_bounding_box = resize_matrix(bounding_box, box_size, box_size)
        for i in range(box_size):
            for j in range(box_size):
                if not test:
                    streched_data[k, box_size*i+j+1] = streched_bounding_box[i, j]
                else:
                    streched_data[k, box_size*i+j] = streched_bounding_box[i,j]
    return streched_data

def resize_matrix(mat, n_row_out, n_col_out):
    n_row_in = mat.shape[0]
    n_col_in = mat.shape[1]
    mat_out = np.empty((n_row_out, n_col_out))
    row_ratio = n_row_in/n_row_out
    col_ratio = n_col_in/n_col_out
    for i in range(n_row_out):
        for j in range(n_col_out):
            mat_out[i, j] = mat[math.floor(i*row_ratio), math.floor(j*col_ratio)]
    return mat_out

def find_top(data, num, original_size):
    for i in range(1, len(data[num, :])):
        if data[num, i].item() is not 0:
            return math.floor(i/original_size)

def find_bottom(data, num, original_size):
    for i in range(len(data[num, :]) - 1, 0, -1):
        if data[num, i].item() is not 0:
            return math.floor(i/original_size)

def find_left(data, num, original_size, test):
    for left in range(original_size):
        for j in range(original_size):
            if not test:
                if data[num, original_size*j+left+1].item() is not 0:
                    return left
            else:
                if data[num, original_size*j+left].item() is not 0:
                    return left

def find_right(data, num, original_size, test):
    for right in range(original_size-1,-1,-1):
        for j in range(original_size):
            if not test:
                if data[num, original_size*j+right+1].item() is not 0:
                    return right
            else:
                if data[num, original_size*j+right].item() is not 0:
                    return right
```