Density Estimation and Classification

Yu Chang
Arizona State University
ASU ID: 1231962201
ychan175@asu.edu

1. INTRODUCTION

The MNIST dataset contains 70,000 images of handwritten digits, divided into 60,000 training images and 10,000 testing images. We use only images for digit "7" and digit "8". We assume that these two features are independent, and that each image is drawn from a 2-D normal distribution. The necessary images are extracted, and stored in “mnist_data.mat” files.

5. You should use trX_new and tsX_new for BOTH naive bayes and logistic regression.

2. FEATURE EXTRACTION

Number of images for digit "7" and digit "8":

1. training set: "7": 6265; "8": 5851.
2. testing set: "7": 1028; "8": 974.

We are required to extract the following two features for each image:

1. The average of all pixel values in the image.
2. The standard deviation of all pixel values in the image.

The dataset has 4 matrices, trX, trY, tsX, tsY. For trX and tsX, each row represents a digit, their dimension is 28 * 28 = 784. The Number 0 represents white digit, 1 represents black digit. The number of rows represents how many digits there are, so the trX has 6265+5851 rows, the tsX has 1028 + 974 rows. The trY and the tsY are labels of the training set. For feature extraction, we calculate the mean and s.d. of each digit.

Extract features:

```
train_data = Numpyfile['trX']
train_y = Numpyfile['trY'][0]
test_data = Numpyfile['tsX']
test_y = Numpyfile['tsY'][0]
```

Using Numpy to calculate:

```
mean_data = np.mean(train_data, axis =1)
std_data = np.std(train_data, axis =1)
```

After the extraction, the new trX and the new tsX have only two columns:

```
temp1 = np.reshape(mean_data, (len(mean_data), 1))
temp2 = np.reshape(std_data, (len(mean_data), 1))
train_data = np.append(temp1, temp2, axis=1)
```

```
temp1 = np.reshape(mean_data_test, (len(mean_data_test), 1))
temp2 = np.reshape(std_data_test, (len(mean_data_test), 1))
test_data = np.append(temp1, temp2, axis=1)
```

3. NAIVE BAYES CLASSIFICATION

1. Initialize

```
    def __init__(self):
        self.mean = {}
        self.covariance = {}
        self.inv_cov = {}
        self.determinant = {}
        self.probs = {}
```

2. Separate trX_new into two sets, one set only trY=0, another trY=1:

```
    def train(self, train, label):
        label_8 = np.transpose(np.argwhere(label == 1))[0]
        label_7 = np.transpose(np.argwhere(label == 0))[0]
        data_8 = train[label_8]
        data_7 = train[label_7]
```

3. Calculate the mean:

```
        mean_8 = np.mean(data_8, axis=0)
        mean_7 = np.mean(data_7, axis=0)
        self.mean[1] = mean_8
        self.mean[0] = mean_7
```

4. Calculate the covariance matrix:

```
        cov_8 = np.cov(np.transpose(data_8))
        cov_7 = np.cov(np.transpose(data_7))
```

5. The covariance matrix is a diagonal matrix, as we are doing naive bayes, the two features should be independent, which means the upper right and lower left element of the matrix should be 0, i.e. cov(X1, X2) = cov(X2, X1) = 0:

```
        cov_8[0][1] = 0
        cov_8[1][0] = 0
        cov_7[0][1] = 0
        cov_7[1][0] = 0
        self.covariance[1] = cov_8
        self.covariance[0] = cov_7
```

6. Calculate the inverse covariance, determinant, and prior probability for later calculate of the probability

```
        self.inv_cov[1] = np.linalg.inv(cov_8)
        self.inv_cov[0] = np.linalg.inv(cov_7)
        self.determinant[1] = np.linalg.det(cov_8)
        self.determinant[0] = np.linalg.det(cov_7)
```

```
        self.probs[1] = len(data_8)/len(train)
        self.probs[0] = len(data_7)/len(train)
```

7. define a function to calculate the probability:

```
    def prob(self, inp, digit):
        temp1 = inp - self.mean[digit]
        temp2 = 0.5*np.dot(np.dot(temp1, self.inv_cov[digit]),
np.transpose(temp1))

        return
self.probs[digit]*np.exp(-temp2)/((2*np.pi)*(self.determinant[digit]**0.5))
```

8. Calculate the probability of the input being 0 and 1 and return which probability is higher:

```
    def predict(self, inp):
        results = []
        for i in range(0, len(inp)):
            if self.prob(inp[i], 0) > self.prob(inp[i], 1):
                results.append(0)
            else:
                results.append(1)

        return results
```

4. LOGISTIC REGRESSION

1. Initialize

```
    def __init__(self):
        self.weight = np.random.random(2)
        self.w0 = 0
        self.lr = 1
        self.no = 10000
```

2. Set activation function:

```
    def sigma(self, inp):
        return 1/(1+np.exp(-inp))
```

3. Calculate the linear combination

```
    def calinp(self, inp):
        temp = np.dot(inp, np.transpose(self.weight))
        return temp+self.w0
```

4. Calculate the loss

```
    def loss(self, inp, y):
        temp = self.sigma(self.calinp(inp))
```

```
        loss = np.dot((y - temp), inp)/len(y)
        return loss, np.sum(y-temp)
```

5. Train the model

```
    def train(self, inp, y):
         for i in range (self.no):
             loss, lossw0 = self.loss(inp, [y])
             self.weight += self.lr*loss[0]
             self.w0 += self.lr*lossw0
             if i % 500 == 0:
                 self.lr /= 2
```

6. Predict the output

```
    def predict(self, inp):
        results = []
        for i in range(0, len(inp)):
            if self.sigma(self.calinp(inp[i])) > 0.5:
                results.append(1)
            else:
                results.append(0)
        return results
```

5. RESULTS

```
mean_data [0.12653061 0.0787565  0.1097489  ... 0.1132453  0.12906162
0.10464186]
std_data [0.30359794 0.24338705 0.28152608 ... 0.27471825 0.29700734
0.26496984]
mean_data_test [0.09230692 0.11338535 0.07626551 ... 0.10408663 0.1277511
0.2267607 ]
std_data_test [0.25874655 0.29069578 0.23425406 ... 0.27412898 0.30596646
0.40250048]
NaiveBayes.mean {1: array([0.15015598, 0.32047584]), 0: array([0.1145277 ,
0.28755657])}
Accuracy using Naive Bayesian for overall, number 7, number 8 as follow:
(0.6953046953046953, 0.7597276264591439, 0.6273100616016427)

Accuracy using Logistic Regression for overall, number 7, number 8 as follow:
(0.8101898101898102, 0.811284046692607, 0.8090349075975359)
```