DENSITY ESTIMATION and CLASSIFICATION

Parameters:

The test and training dataset are represented by “trX” and “tsX” respectively. Their corresponding labels or targets are stored in “trY” and “tsY” respectively.

Naïve Bayes Classification:

Naive Bayes classifier is a probabilistic classifier based on Bayes' theorem, in which the features of the classes are independent of each other. The probability that an image belongs to a specific category is expressed as:

P (Y=Class / X) = P (X/Y=Class) * P (Y= Class) / P(X)

Every element

P (Y=Class/X) posterior,

P (X/Y=Class) likelihood,

P (Y= Class) Class Prior Probability

P(X) Predictor Prior Probability.

However, the condition of Naive Bayes is that all functions are independent. We have two characteristics, namely mean and standard deviation. Therefore, we can write the equation as

P (Y=Class/X) = P (Class/X1) * P (Class/X2)

where X1 is Mean

X2 is Standard Deviation

In the code I write like this, it follows Normal distribution

```
def p_x_given_y(x, mean, variance):
    p_x_give_y = (1 / (np.sqrt(2 * np.pi * variance))) * np.exp(-(x - mean) ** 2 / (2 * variance))
    return p_x_give_y
```

Finally, I write the posterior probability of class 7 and class 8 as

```
post_prob_7 = prob_7 \
              * p_x_given_y(testing_set_mean, mean_7.mean(), mean_7.var()) \
              * p_x_given_y(testing_set_sd, sd_7.mean(), sd_7.var())

# Calculating the 'numerator' for posterior probability for 8
post_prob_8 = prob_8 \
              * p_x_given_y(testing_set_mean, mean_8.mean(), mean_8.var()) \
              * p_x_given_y(testing_set_sd, sd_8.mean(), sd_8.var())
```

I calculate the posterior probability numerators of categories 7 and 8 and compare these values. If the posterior probability numerator of category 7 is greater than that of category 8, the image is classified as belonging to category -7, and vice versa

```
value_compare = np.greater(post_prob_8, post_prob_7)
```

Logistic Regression Classification:

The hypothesis is :

```
def sigmoid_func(x):
    return 1 / (1 + np.exp(-x))
```

$$h_\theta(\mathbf{x}) = g(\theta^\top \mathbf{x})$$

$$g(z) = \frac{1}{1+e^{-z}}$$

The parameter of the logic function is θ, which is the coefficient or weight. We need to calculate the parameter θ to maximize the log-likelihood

$$\mathbf{w}* = \text{argmax}\mathbf{w}l(\theta)$$

$$\mathbf{w}* = \text{argmax}\theta \sum [y(i)\theta tx(i) - \log\Phi 1 + \exp(\theta tx(i))\Phi] \; n \; i{=}1$$

I will use Gradient Ascent to maximize the equation. Gradient rising equation used to find θ by

$\theta(k{+}1) = \theta(k) + \eta\nabla\theta(k)l(\theta)$

$\eta > 0$ is called the learning rate

l(θ) represents the log likelihood function

```
def training_gradient_ascent(X, y, learning_rate, iterations):

    # initialize weights
    total_sample, total_features = X.shape
    weights = np.zeros(total_features)
    # gradient ascent
    for iterations in range(iterations):
        linear_comb = np.dot(X, weights)
        # apply sigmoid_func function
        y_predicted = sigmoid_func(linear_comb)
        # calculate gradient
        gradient = (1 / total_sample) * np.dot(X.T, (y - y_predicted))
        # update weights
        weights += learning_rate * gradient

    return weights
```

I set up a function here to calculate the accuracy

```
def calculate_accuracy (compared, true_labels, test_for):

    if test_for == 0:
        # testing for hand written 7
        # squeeze the input array
        compared_array = np.squeeze(compared)[0:1028]
        true_labels_array = np.squeeze(true_labels)[0:1028]

    elif test_for == 1:
        compared_array = np.squeeze(compared)[1028:]
        true_labels_array = np.squeeze(true_labels)[1028:]
    else:
        compared_array = np.squeeze(compared)
        true_labels_array = np.squeeze(true_labels)

    total_test_case = true_labels_array.shape[0]

    test_result = np.equal(compared_array, true_labels_array)
    # now count the positive cases
    true_cases = np.count_nonzero(test_result)

    accuracy = true_cases/total_test_case
    accuracy_percent = accuracy * 100

    return accuracy, accuracy_percent
```

We set the number of iterations to 10,000 and the learning rate to 0.001.

```
#setup learning rate and max iteration for LR
learning_rate = 0.001
max_iteration = 10000
```

If the result of the sigmoid function is greater than 0.5, it is classified into class 8 , Otherwise class 7.

Result:

```
The Accuracy of the Naive Bayes for predicting "7" is  75.9727626459144 %
The Accuracy of the Naive Bayes for predicting "8" is  62.73100616016427 %
The Accuracy of the Naive Bayes is  69.53046953046953 %
The Accuracy of the Logistic Regression for predicting "7" is  96.10894941634241 %
The Accuracy of the Logistic Regression for predicting "8" is  97.5359342915811 %
The Accuracy of the Logistic Regression is  96.8031968031968 %
Learning Rate has set to  0.001
Max iteration has set to  10000
```