Report of Data Labeling using Linear Classifiers

Xuan ZHOU
Institut Polytechnique de Parise
xuan.zhou@telecom-sudparis.eu

Part I. Nearest Neighbor classifier and K- Nearest Neighbor classifier

This lab is to design a simple linear classifier capable of automatically predicting the labels of unseen data as input to the system. In Part I, learning nearest neighbor classification and K-nearest neighbor classification, and in Part II, learning multiclass support vector machine loss data classification gives us a fuller understanding of the basic linear classifier concepts. In the nearest neighbor classifier, we will apply the nearest neighbor classifier on a popular image database called CIFAR-10. This dataset consists of 60.000 small images with a height and width of 32 pixels. Each image is labeled as one of 10 categories. These 60,000 images are divided into a training set of 50,000 images and a test set of 10,000 images. The nearest neighbor classifier takes a test image, compares it to each training image, and predicts the label of the closest training image.

We design a nearest neighbor classifier able to predict the category of any image in the test dataset, given two images and representing them as vectors I_1 and I_2, a naive choice for comparing them might be the L_1 distance:

$$d_1(I_1, I_2) = \sum_p |I_1^p - I_2^p|$$

First, the input data/labels of CIFAR-10 are loaded into Python and the results of generating the sizes of the four vectors (x_train, y_train, x_test, y_test) are shown in Figure 1.

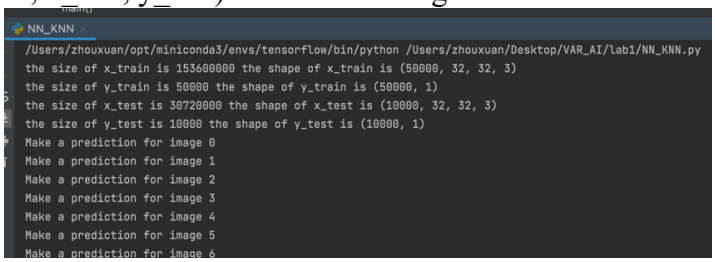


Figure 1. Load the CIFAR-10 data and generate four vectors

The results of visualizing the top 10 images in the test dataset with their associated labels are shown in Figure 2.

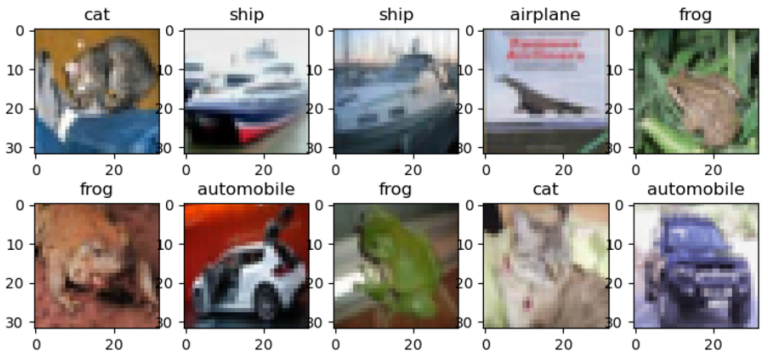


Figure 2. Visualizing the top 10 images in the test dataset

To calculate the difference between the two images, the data of the nearest neighbor classifier is flattened and the reshaped training and test sets are reshaped from a matrix structure to a vector of floating points. This experiment predicts the labels of the top 200 images present in the test data. During the training process, each

image of the training dataset and the associated labels are first to read. Second, the absolute difference between the images in the training set and the query images is calculated. Third, all the differences are summed to get a score. Fourth, the most recently received image in the training dataset is identified and its associated label is kept. The predicted labels of the query images are compared with the ground truth labels, and the number of correct predicted images is increased if the prediction is correct, and the result is shown in Figure 3.

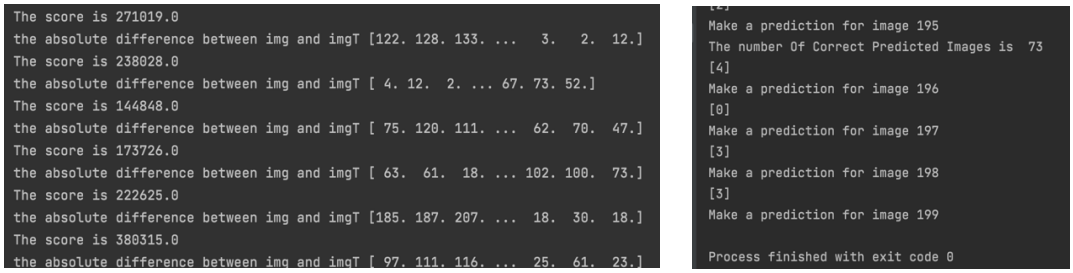


Figure 3. Results of reshaping the Nearest Neighbor classifier

In order to calculate the accuracy of the predicted labels precisely, we use the following calculation method.

$$Accuracy = 100 * \frac{Nr\_of\_correct\_prediction}{No\_of\_images\_considered\_for\_testing}$$

The accuracy result of the Nearest Neighbor classifier is 36.5%, which is shown in Figure 4.

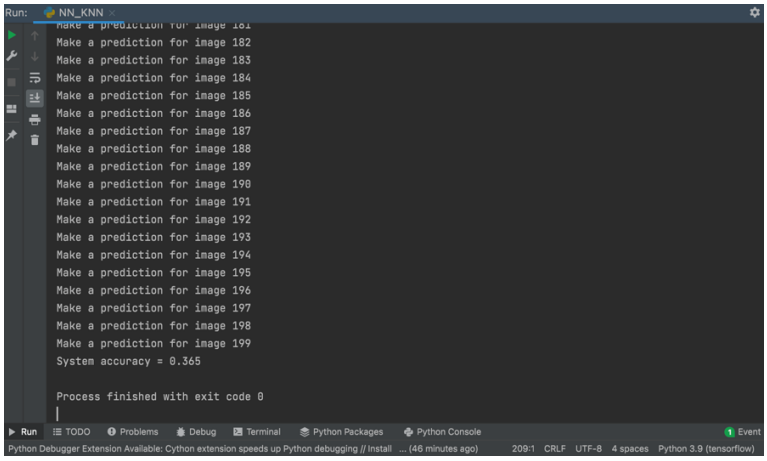


Figure 4. The accuracy results of the Nearest Neighbor classifier

Next, we design a K- Nearest Neighbor classifier to predict the class of any image in the test set data. In the K- Nearest Neighbor classifier, it will compare the images pixel by pixel and add up all the differences, given two images and representing them as vectors I_1 and I_2, a naive choice for comparing them might be the L_1 distance:

$$d_1(I_1, I_2) = \sum_p \left| I_1^p - I_2^p \right|$$

With this formula, the two images are subtracted by element and all differences are added as a score. The system will determine the top k images based on the similarity score and determine the image labels based on the maximum consensus between the returned categories. The first input is the reshaped training dataset, the labels of the training dataset and the test images. The output of the function should be the predicted label for the current input, i.e., the predicted_Label is returned. in the predict_Label_KNN function, first, each image in the training dataset and its associated label is read. Second, the absolute difference between the images in the training dataset and the test image is calculated and the differences of all pixels are summed to obtain a score. Third, all the scores and associated labels are stored in a list, and all the elements in the prediction list are arranged in ascending order of priority according to the scores. Fourth, the first k=10

predicted values of the current image are kept, and then only the first K predicted labels are extracted in a separate vector. The result is shown in Figure 5.

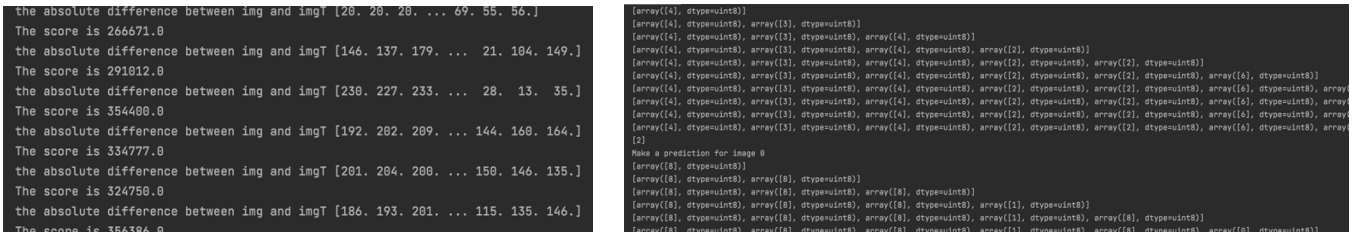


Figure 5. The Results of reshaping the K- Nearest Neighbor classifier

The accuracy result of the K-Nearest Neighbor classifier is78%, which is shown in Figure 6.

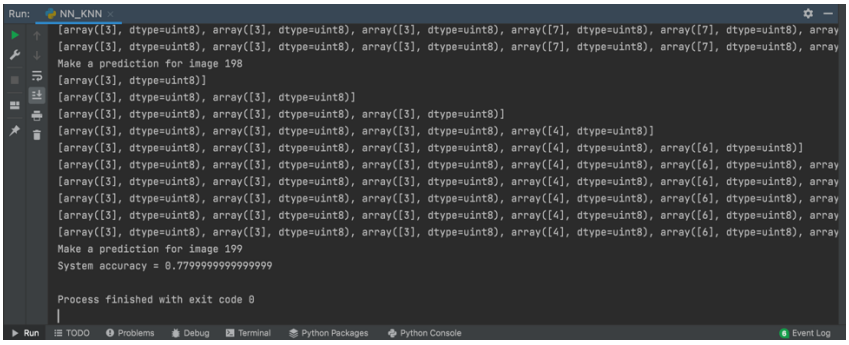


Figure 6. The accuracy results of the K-Nearest Neighbor classifier

In addition, we need to discuss the use of L_2 distance to calculate the Euclidean distance between two vectors. That is, after we calculate the difference between the pixels, we choose to add all the pixels squared and finally take the square root. The calculation formula is shown below.

$$d_2(I_1, I_2) = \sqrt{\sum_p \left(I_1^p - I_2^p\right)^2}$$

The results of using the distance calculation formula for L_2 are shown in Figure 7.

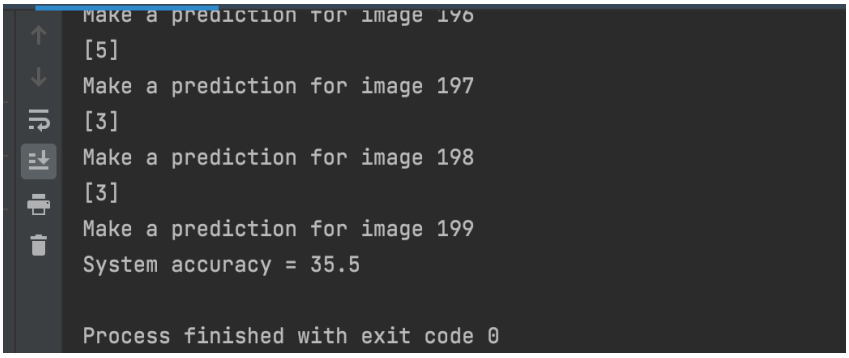


Figure 7. The accuracy results of L2 distance

During the run, the code takes longer to run than the L_1 distance formula. This is due to the fact that the L_2 distance formula is very computationally intensive in terms of calculating the difference between pixels and taking the square root at the end. And the final accuracy is 35.5%

Next, we modified the number of neighbors of the k-NN algorithm to be applied to the Cifar10 dataset for system performance evaluation for different numbers of neighbors. KNN uses proximity to classify or predict groups of individual data points. That is, given a training dataset, for a new input instance, find the K instances in the training dataset that are closest to the instance, and the majority of these K instances belong to a certain class, and classify that input instance into that class. In this experiment, I chose prediction values of 3, 5, 10, 20, and 50 to make predictions. The results of each prediction are shown in Figure 8. Figure 9

shows the details of the results for each prediction. Figure 10 shows that in order to determine the similarity of the images given L_2 distance, I chose 3, 5, 10, 20 and 50 prediction values to make predictions. Figure 11 shows the details of the Similarity of images at L_2 distance.

No of neighbors	1	3	5	10	20	50
System accuracy	36.5	35.5	38.0	39.0	38.5	35.0
Time costing	55.31 s	56.80 s	56.87 s	56.89 s	57.96 s	57.46 s

Figure 8. System performance evaluation for various numbers of neighbors

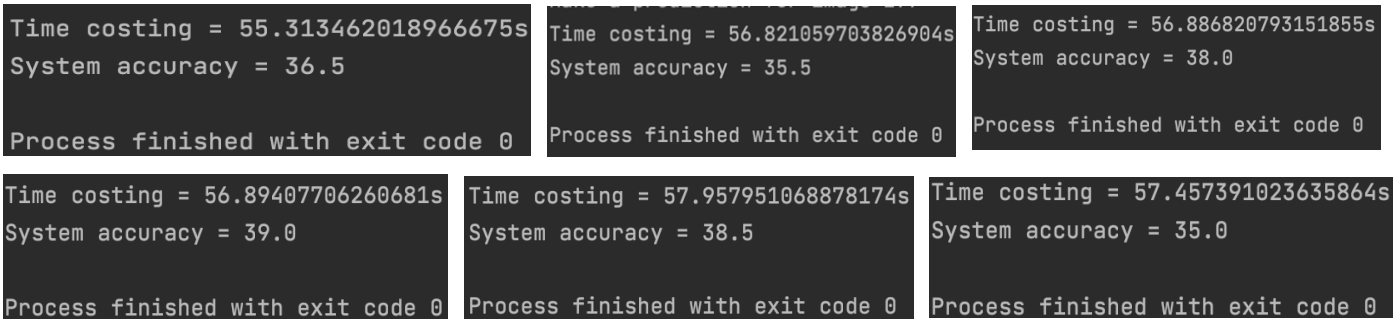


Figure 9. The details of the results for each prediction

No of neighbors	1	3	5	10	20	50
System accuracy	35.5	34.5	36.0	38.0	35.5	35.5
Time costing	332.35 s	333.77 s	335.18 s	334.70	331.99 s	335.64 s

Figure 10. System performance evaluation for various numbers of neighbors

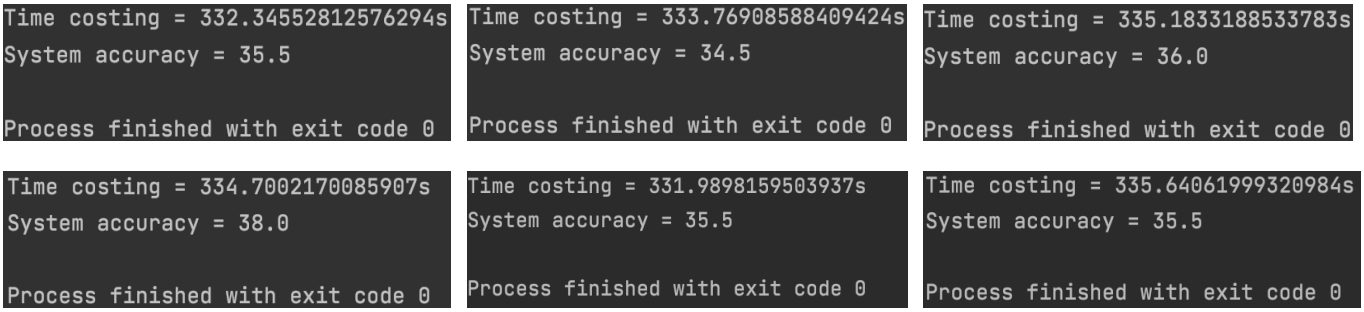


Figure 11. The details of the Similarity of images at L_2 distance

In the L_1 algorithm, the system accuracy of KNN is 36.5% when K is equal to 1. When in the L_2 algorithm, the system accuracy of KNN is 35.5% when K is equal to 1. The system accuracy is equal to that of the Nearest Neighbor classifier when K is equal to 1 in both L_1 and L_2 algorithms.

Part II. Linear Classifier with SVM Loss

In the second part, we will design a linear classifier that can be tuned using the SVM loss function. The set of weights W is first randomly initialized to predict the correct label of the dataset (x_i) used as input. In this a priori classifier, a dataset with six points in a four-dimensional space is [1, 5, 1, 4], [2, 4, 0, 3], [2, 1, 3, 3], [2, 0, 4, 2], [5, 1, 0, 2], [4, 2, 1, 1], which is labeled [0, 0, 1, 1, 2, 2]. The test set is [1, 5, 2, 4], [2, 1, 2, 3], [4, 1, 0, 1], which is labeled [0, 1, 2]. and a set of weights stored in the matrix ($\boldsymbol{W}$) will be used as input. The system will modify the elements of the $\boldsymbol{W}$ matrix so that the prediction ($\boldsymbol{W} \cdot x_i$) is consistent with the ground truth label set (y_train). Finally, after the training, a new set of points (x_test) will be used as input and the system will predict the category for them. First load the training set, test set and weight matrix. A function is defined in the calculation of the score, which takes two parameters, training data points and a weight matrix. And as output, a score vector is returned. For the 1st data sample, the SVM loss takes the following form:

$$L_i = \sum_{j \neq y_i} max(0, s_j - s_{y_i} + \Delta)$$

After running, when this epoch is 1, the global normalized loss is 3.83, as shown in Figure 12.

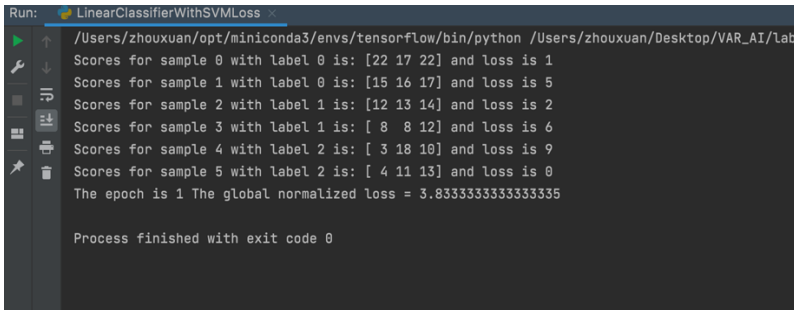


Figure 12. The result of adjusting the weights matrix

Next, the change in the loss function is calculated to be less than 0.001 by changing the setting of the weights in exercise 6. I set the weights to 1000. as shown in Figure 13, the global normalized loss is less than 0.001 when epoch is 159, which is 0.00032460269018046406.

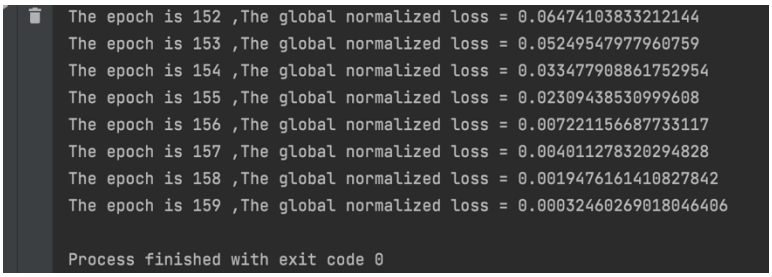


Figure 13. The number of steps necessary for the algorithm to converge

Based on the set of weights determined in exercise 6, the labels of all points present in the x_test variable are predicted. The results are compared with the ground truth labels (y_test) and an accuracy of 100% is obtained. As shown in Figure 14.

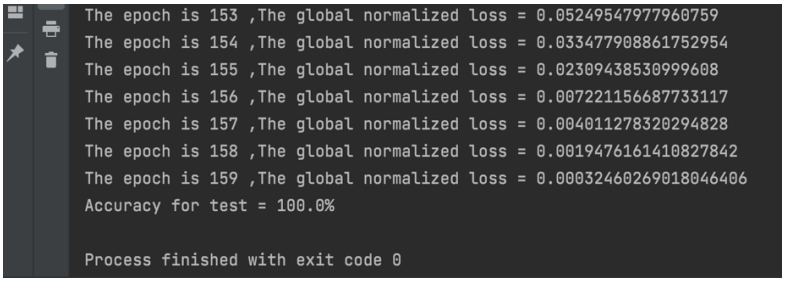


Figure 14. The accuracy of the labels of all points in the x_test

The last experiment is about making predictions on the iris dataset. This is a multi-category classification problem. The number of observations in each category is balanced. There are 150 observations with 4 input variables and 1 output variable. As shown in Figure 15.

SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm	Class
5.1	3.5	1.4	0.2	Iris-setosa
4.9	3	1.4	0.2	Iris-setosa
4.7	3.2	1.3	0.2	Iris-setosa
4.6	3.1	1.5	0.2	Iris-setosa
5	3.6	1.4	0.2	Iris-setosa
5.4	3.9	1.7	0.4	Iris-setosa
4.6	3.4	1.4	0.3	Iris-setosa
5	3.4	1.5	0.2	Iris-setosa
4.4	2.9	1.4	0.2	Iris-setosa
4.9	3.1	1.5	0.1	Iris-setosa
5.4	3.7	1.5	0.2	Iris-setosa
4.8	3.4	1.6	0.2	Iris-setosa
4.8	3	1.4	0.1	Iris-setosa
4.3	3	1.1	0.1	Iris-setosa
5.8	4	1.2	0.2	Iris-setosa

Figure 15. The dataset of the iris

Of the 150 observations of Iris, 120 elements are used for training and 30 elements are used for testing. The weight matrix W will be initialized as a random number from 0 to 1. specified after performing system training/evaluation. To obtain maximum testing accuracy, adjust the weights without dropping steps to the optimal value of thisdepends on the learning rate.In this experiment, I set the learning rate to the 11 numbers 1, 0.5, 0.1, 0.05, 0.01, 0.005, 0.001, 0.0005, 0.0001, 0.00005, 0.00001. As shown in Figure 16, when the learning rate is 1, the maximum test accuracy is 100%, and at this time the optimal value for the weights is 200. When the learning rate is 0.1, the maximum test accuracy is 100%, and the optimal value for the weights is 374. When the learning rate is 0.05, the maximum test accuracy is 100%, and the optimal value for the weights is 374. When the learning rate is 0.01, the maximum test accuracy is 100%, and the optimal value for the weights is 238. When the learning rate is 0.005, the maximum test accuracy is 100%, and the optimal value for the weights is 371. When the learning rate is 0.001, the maximum test accuracy is 100%, and the optimal value for the weights is 2406. When the learning rate is 0.0001, the maximum test accuracy is 96.55%, and the optimal value for the weights is 5937. When the learning rate is 0.00005, the maximum test accuracy is 72.41%, and the optimal value for the weights is 9929. When the learning rate is 0.00005, the maximum test accuracy is 51.72%, and the optimal value for the weights is 8459.

```
Learning Rate = 1 The Maximum accuracy is 100.0% , The optimal step value is 200
Learning Rate = 0.5 The Maximum accuracy is 100.0% , The optimal step value is 208
Learning Rate = 0.1 The Maximum accuracy is 100.0% , The optimal step value is 374
Learning Rate = 0.05 The Maximum accuracy is 100.0% , The optimal step value is 371
Learning Rate = 0.01 The Maximum accuracy is 100.0% , The optimal step value is 238
Learning Rate = 0.005 The Maximum accuracy is 100.0% , The optimal step value is 481
Learning Rate = 0.001 The Maximum accuracy is 100.0% , The optimal step value is 2406
Learning Rate = 0.0005 The Maximum accuracy is 100.0% , The optimal step value is 4811
Learning Rate = 0.0001 The Maximum accuracy is 96.55172413793103% , The optimal step value is 5937
Learning Rate = 5e-05 The Maximum accuracy is 72.41379310344827% , The optimal step value is 9929
Learning Rate = 1e-05 The Maximum accuracy is 51.724137931034484% , The optimal step value is 8459

Process finished with exit code 0
```

Figure 16. The result of the optimal value for the weights

In order to obtain an accuracy rate higher than 90%, I also set different learning rates of 1, 0.5, 0.1, 0.05, 0.01, 0.005, 0.001 and 0.0005. When the learning rate is 1, the current maximum accuracy is 96.55% and the minimum number of steps is 24. When the learning rate is 0.5, the current maximum accuracy is 96.55% and the minimum number of steps is 24. When the learning rate is 0.1, the current maximum accuracy is 100%, and the minimum number of steps is 31. When the learning rate is 0.05, the current maximum accuracy is 93.10%, and the minimum number of steps is 24. When the learning rate is 0.01, the current maximum accuracy is 93.10%, and the minimum number of steps is 246. When the learning rate is 0.005, the current maximum accuracy is 93.10%, and the minimum number of steps is 24. The maximum accuracy varies for different learning rates, and the minimum number of steps is also different, as shown in Figure 17.

```
/Users/zhouxuan/opt/miniconda3/envs/tensorflow/bin/python /Users/zhouxuan/Desktop/VAR_AI/lab1/Iris.py
Learning Rate = 1 The Maximum accuracy is 96.55172413793103% , The optimal step value is 24
Learning Rate = 0.5 The Maximum accuracy is 96.55172413793103% , The optimal step value is 20
Learning Rate = 0.1 The Maximum accuracy is 100.0% , The optimal step value is 31
Learning Rate = 0.05 The Maximum accuracy is 93.10344827586206% , The optimal step value is 115
Learning Rate = 0.01 The Maximum accuracy is 93.10344827586206% , The optimal step value is 246
Learning Rate = 0.005 The Maximum accuracy is 93.10344827586206% , The optimal step value is 499
Learning Rate = 0.001 The Maximum accuracy is 93.10344827586206% , The optimal step value is 2510
Learning Rate = 0.0005 The Maximum accuracy is 93.10344827586206% , The optimal step value is 5015

Process finished with exit code 0
```

Figure 17. The minimum number of steps necessary

The system does suffer from random initialization of the weight matrix during the training process. In this experiment, I set the weights to a matrix of all zeros, i.e., [0,0,0,0,0],[0,0,0,0],[0,0,0,0], a matrix of all ones, which is [1, 1, 1, 1], [1, 1, 1, 1], [1, 1, 1, 1], a matrix of all 100s, which is[100, 100, 100, 100],[100, 100, 100, 100],[100, 100, 100, 100] and three random matrices with a fixed learning rate of 0.01. The accuracy of different weight matrices, the optimal value and the lowest loss in the training set are different and they have different learning curves. Figure 18 shows the accuracy curve and the loss function curve of the all-0 weight matrix. Figure 19 shows the accuracy curve and the loss function curve of the full 1 weight matrix. Figure 20 shows the accuracy curve and the loss function curve of the full 100-weight matrix. Figure 21 shows the accuracy curve and loss function curve of the random weight matrix.

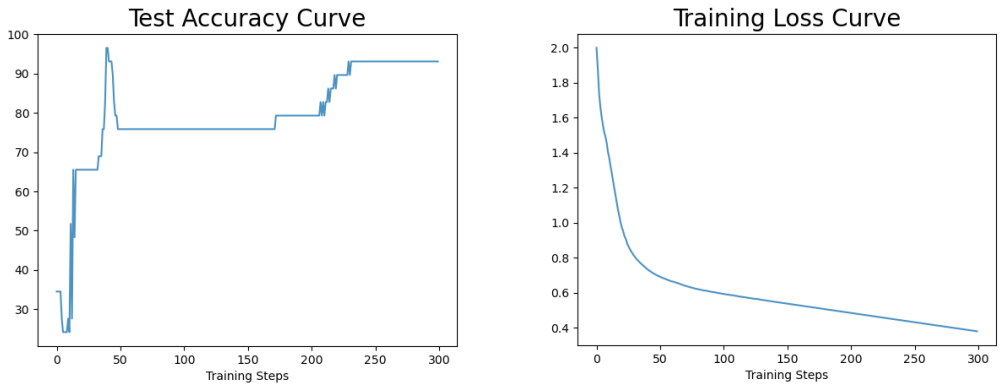


Figure 18. The accuracy curve and the loss function curve of the all-0 weight matrix

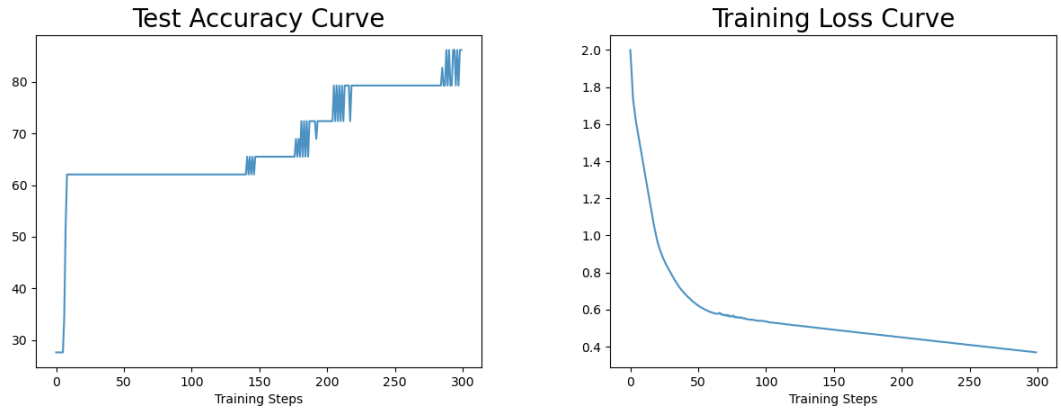


Figure 19. The accuracy curve and the loss function curve of the all-1 weight matrix

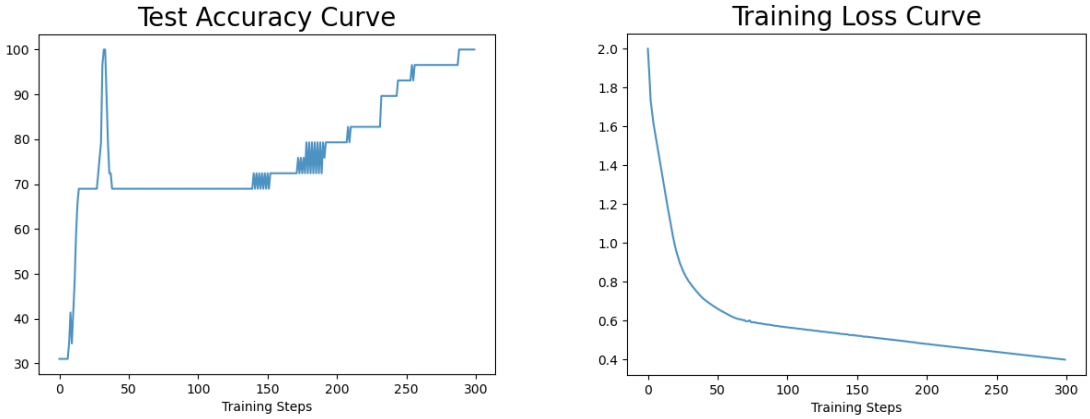
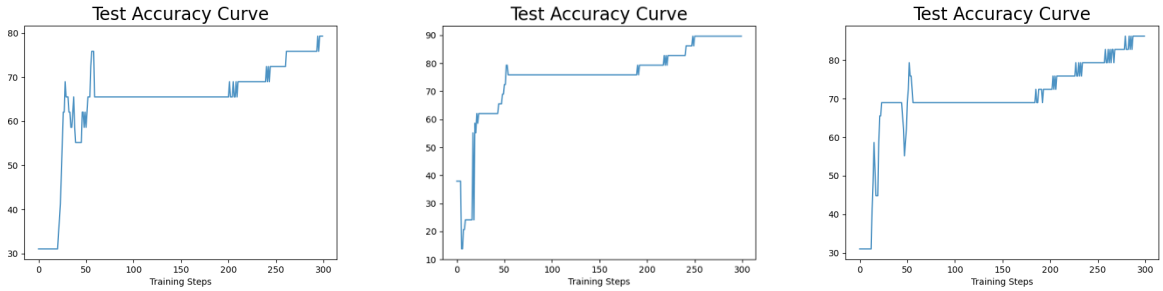


Figure 20. The accuracy curve and the loss function curve of the all-100 weight matrix



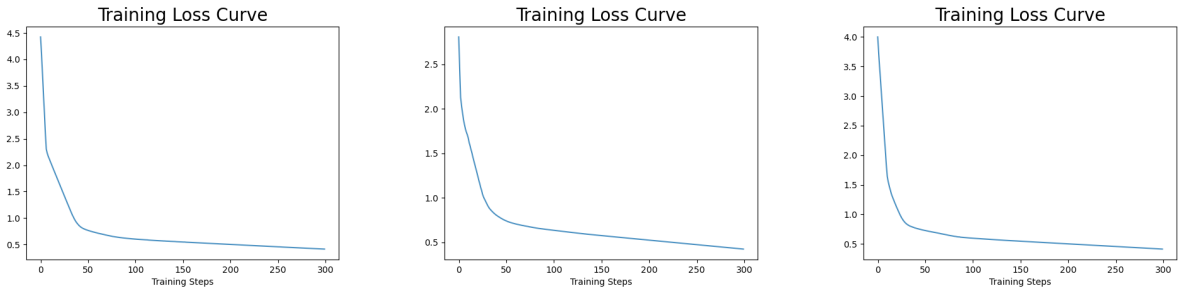


Figure 21. The accuracy curve and the loss function curve of the random weight matrix

When the weight matrix is an all-0 matrix, the maximum accuracy is 96.55%, the optimal step value is 40 and the lowest loss in training set is 0.38. When the weight matrix is an all-1 matrix, the maximum accuracy is 86.21%, the optimal step value is 289 and the lowest loss in training set is 0.37. When the weight matrix is a full 100 matrix, the maximum accuracy is 100%, the optimal step value is 33 and the lowest loss in training set is 0.40. When the weight matrix is a random matrix, the accuracy and the optimal step value and the lowest loss in training set are shown in Figure 22 below.

```
/Users/zhouxuan/opt/miniconda3/envs/tensorflow/bin/python /Users/zhouxuan/Desktop/VAR_AI/lab1/Iris.py
Learning Rate = 0.01 The Maximum accuracy is 96.55172413793103% , The optimal step value is 40 The lowest Loss in training set = 0.3790887503433204

Process finished with exit code 0
```

```
Iris
/Users/zhouxuan/opt/miniconda3/envs/tensorflow/bin/python /Users/zhouxuan/Desktop/VAR_AI/lab1/Iris.py
Learning Rate = 0.01 The Maximum accuracy is 86.20689655172413% , The optimal step value is 289 The lowest Loss in training set = 0.3692033224276737

Process finished with exit code 0
```

```
Iris
/Users/zhouxuan/opt/miniconda3/envs/tensorflow/bin/python /Users/zhouxuan/Desktop/VAR_AI/lab1/Iris.py
Learning Rate = 0.01 The Maximum accuracy is 100.0% , The optimal step value is 33 The lowest Loss in training set = 0.39869489825305043

Process finished with exit code 0
```

```
/Users/zhouxuan/opt/miniconda3/envs/tensorflow/bin/python /Users/zhouxuan/Desktop/VAR_AI/lab1/Iris.py
Learning Rate = 0.01 The Maximum accuracy is 79.3103448275862% , The optimal step value is 295 The lowest Loss in training set = 0.41426948225286053

Process finished with exit code 0
```

```
/Users/zhouxuan/opt/miniconda3/envs/tensorflow/bin/python /Users/zhouxuan/Desktop/VAR_AI/lab1/Iris.py
Learning Rate = 0.01 The Maximum accuracy is 89.65517241379311% , The optimal step value is 249 The lowest Loss in training set = 0.42309227958104834

Process finished with exit code 0
```

```
/Users/zhouxuan/opt/miniconda3/envs/tensorflow/bin/python /Users/zhouxuan/Desktop/VAR_AI/lab1/Iris.py
Learning Rate = 0.01 The Maximum accuracy is 93.10344827586206% , The optimal step value is 45 The lowest Loss in training set = 0.35328739809466864

Process finished with exit code 0
```

Figure 22. Effect of random initialization of the weight matrix

We can conclude that the system does suffer from the random initialization of the weight matrix during the training process. And this system is able to achieve 100% accuracy in the testing phase. It also achieves 100% accuracy in different stages of the test, as shown in Figure 23.

```
The epoch is 159 ,The global normalized loss = 0.00032460269018046406
Accuracy for test = 100.0%

Process finished with exit code 0
```

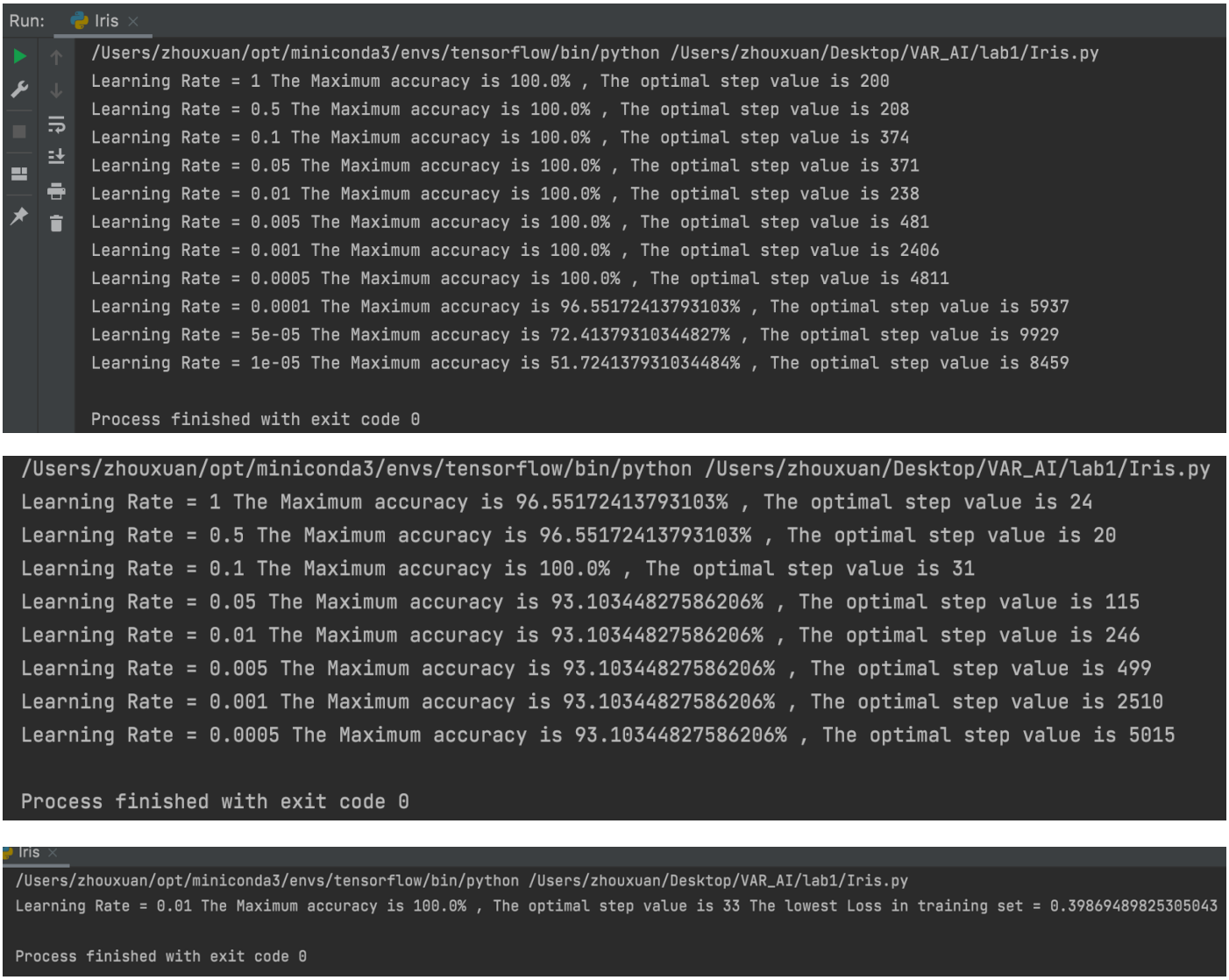


Figure 22. The results of 100% accuracy in this system

In this experiment, I learned about the Nearest Neighbor classifier and K- Nearest Neighbor classifier, and also understood the role of the Linear Classifier in SVM Loss through the experiment. During this experiment, the step-by-step guide steps allowed me to go deeper into the knowledge as well, which was perfect for me. I really enjoyed this guided experiment, which made me enjoy it.

Appendix

Code for Exercise 1

```
print("the size of x_train is", x_train.size, "the shape of x_train is", x_train.shape)
print("the size of y_train is", y_train.size, "the shape of y_train is", y_train.shape)
print("the size of x_test is", x_test.size, "the shape of x_test is", x_test.shape)
print("the size of y_test is", y_test.size, "the shape of y_test is", y_test.shape)
```

Comment: Use the function *.size to know the size of the matrix, and use the function *.shape to know the shape of the matrix.

Code for Exercise 2

```
cols = 5
rows = 2
fig = plt.figure(figsize=(2 * cols - 1, 2.5 * rows - 1))
idx=0
for i in range(cols):
    for j in range(rows):
        ax = fig.add_subplot(rows, cols, i * rows + j + 1)
        ax.imshow(x_test[idx])
        label = y_test[idx]
        print(label)
        ax.set_title(dict_classes[y_test[idx, 0]])
        idx = idx + 1
    plt.show()
```

Comment: Determine the rows and columns of the printed images, and visualize the images in the dataset through a two-level loop.

Code for Exercise 3

```
#exercise3
import numpy as np
import cv2
import tensorflow as tf
import matplotlib
matplotlib.use('TkAgg')
import matplotlib.pyplot as plt

dict_classes = {0: "airplane", 1: "automobile", 2: "bird", 3: "cat", 4: "deer", 5: "dog", 6: "frog", 7: "horse", 8: "ship", 9: "truck"}

def most_frequent(list):
    list = [x[0] for x in list]
    return [max(set(list), key = list.count)]
```

```
def predictLabelNN(x_train_flatten, y_train, img):
    predictedLabel = -1
    scoreMin = 100000000
    for idx, imgT in enumerate(x_train_flatten):
        difference = np.power((img - imgT), 2)
        score = np.sqrt(np.sum(difference))
        if score < scoreMin:
            scoreMin = score
            predictedLabel = y_train[idx]
    return predictedLabel
    def main():
        (x_train,y_train),(x_test,y_test)=tf.keras.datasets.cifar10.load_data()
        '''Exercise 1 - Determine the size of the four vectors x_train, y_train, x_test, y_test'''
        # TODO - Exercise 1 - Determine the size of the four vectors x_train, y_train, x_test, y_test
        #print("the size of x_train is", x_train.size, "the shape of x_train is", x_train.shape)
        #print("the size of y_train is", y_train.size, "the shape of y_train is", y_train.shape)
        #print("the size of x_test is", x_test.size, "the shape of x_test is", x_test.shape)
        #print("the size of y_test is", y_test.size, "the shape of y_test is", y_test.shape)
        '''Exercise 2 - Visualize the first 10 images from the testing dataset with the associated labels'''
        '''cols = 5
        rows = 2
        fig = plt.figure(figsize=(2 * cols - 1, 2.5 * rows - 1))
        idx=0
        for i in range(cols):
            for j in range(rows):
                ax = fig.add_subplot(rows, cols, i * rows + j + 1)
                ax.imshow(x_test[idx])
                label = y_test[idx]
                print(label)
                ax.set_title(dict_classes[y_test[idx, 0]])
                idx = idx + 1
        plt.show()'''
        x_train_flatten = np.float64(x_train.reshape(x_train.shape[0], 32 * 32 * 3))
        x_test_flatten = np.float64(x_test.reshape(x_test.shape[0], 32 * 32 * 3))
        numberOfCorrectPredictedImages = 0
        # Exercise 3
        for idx, img in enumerate(x_test_flatten[0:200]):
            predictedLabelNN = predictLabelNN(x_train_flatten,y_train,img)
            print("Make a prediction for image {}".format(idx))
            if predictedLabelNN == y_test[idx]:
                numberOfCorrectPredictedImages = numberOfCorrectPredictedImages + 1
        accuracy = numberOfCorrectPredictedImages / len(y_test) *100
        print("System accuracy = {}".format(accuracy))
        return
```

Comment: By defining the function predictLabelNN(x_train_flatten, y_train, img) to calculate the difference between the training image and the test image, then square it, and finally open the root to calculate the L2 distance between a test image and other images, by comparing the current distance score and the

predetermined score, get the smallest distance score as The minimum score, and then use the minimum distance as the prediction value to find the corresponding label. By calling the labels in the main function, comparing the predicted labels with the real labels, getting the total number of correct predictions, and finally dividing by the total number of predictions, we can get the exact value.

Code for Exercise 4

```
import numpy as np
import cv2
import tensorflow as tf
import matplotlib
import time
matplotlib.use('TkAgg')
import matplotlib.pyplot as plt

dict_classes = {0: "airplane", 1: "automobile", 2: "bird", 3: "cat", 4: "deer", 5: "dog", 6: "frog", 7: "horse", 8: "ship", 9: "truck"}

# Exercise 4
def most_frequent(list):
    list = [x[0] for x in list]
    return [max(set(list), key = list.count)]

def predictLabeKNN(x_train_flatten, y_train, img):
    predictedLabel = -1
    scoreMin = 100000000
    for idx, imgT in enumerate(x_train_flatten):
        diff = img-imgT
        difference = abs(diff)
        score = np.sum(difference)
        if score < scoreMin:
            scoreMin = score
            predictedLabel = y_train[idx]
    return predictedLabel


def main():
    (x_train,y_train),(x_test,y_test)=tf.keras.datasets.cifar10.load_data()
    #Exercise 1 - Determine the size of the four vectors x_train, y_train, x_test, y_test'"
    # TODO - Exercise 1 - Determine the size of the four vectors x_train, y_train, x_test, y_test
    #print("the size of x_train is", x_train.size, "the shape of x_train is", x_train.shape)
    #print("the size of y_train is", y_train.size, "the shape of y_train is", y_train.shape)
    #print("the size of x_test is", x_test.size, "the shape of x_test is", x_test.shape)
    #print("the size of y_test is", y_test.size, "the shape of y_test is", y_test.shape)
    '"Exercise 2 - Visualize the first 10 images from the testing dataset with the associated labels'"
    '"cols = 5
    rows = 2
    fig = plt.figure(figsize=(2 * cols - 1, 2.5 * rows - 1))
```

```
    idx=0
    for i in range(cols):
        for j in range(rows):
            ax = fig.add_subplot(rows, cols, i * rows + j + 1)
            ax.imshow(x_test[idx])
            label = y_test[idx]
            print(label)
            ax.set_title(dict_classes[y_test[idx, 0]])
            idx = idx + 1
    plt.show()'"
    # Exercise 4
    x_train_flatten = np.float64(x_train.reshape(x_train.shape[0], 32 * 32 * 3))
    x_test_flatten = np.float64(x_test.reshape(x_test.shape[0], 32 * 32 * 3))
    numberOfCorrectPredictedImages = 0
    time_begin = time.time()
    for idx, img in enumerate(x_test_flatten[0:200]):
        print("Make a prediction for image {}".format(idx))
        if predictedLabelKNN == y_test[idx]:
            numberOfCorrectPredictedImages = numberOfCorrectPredictedImages + 1
    time_end = time.time()
    print("Time costing = {}s".format(time_end-time_begin))
    accuracy = numberOfCorrectPredictedImages / len(y_test[0:200]) *100
    print("System accuracy = {}".format(accuracy))
    return
```

Comment: The KNN algorithm uses L2 distance to get the minimum score by calculating the difference between images. The minimum score is found by comparing it with the score initially set by the function. Then the number of correct predictions is divided by the total number of predictions to get the exact value, as shown in Figure 8 and Figure 9.

Code for Exercise 5

```
def predictLabeKNN(x_train_flatten, y_train, img):
    predictedLabel = -1
    scoreMin = 100000000
    for idx, imgT in enumerate(x_train_flatten):
        diff = img-imgT
        difference = np.power(diff, 2)
        score = np.sqrt(np.sum(difference))
        if score < scoreMin:
            scoreMin = score
            predictedLabel = y_train[idx]
    return predictedLabel
```

Comment: Using L_2 distance to measure the distance of feature points, as shown by Figure 10 and Figure 11.

Code for Exercise 6 & 7

```
import numpy as np

#Exercise 6
def predict(xsample, W):
    s = []
    s = np.dot(W, xsample)
    return s

def computeLossForASample(s, labelForSample, delta):
    loss_i = 0
    syi = s[labelForSample]
    for i in range(3):
        if i == labelForSample:
            continue
        loss_i = max(0, s[i] - syi + delta) + loss_i
    return loss_i

def computeLossGradientForASample(W, s, currentDataPoint, labelForSample, delta):
    dW_i = np.zeros(W.shape)
    syi = s[labelForSample]
    for j, sj in enumerate(s):
        dist = sj - syi + delta
        if j == labelForSample:
            continue
        if dist > 0:
            dW_i[j] = currentDataPoint
            dW_i[labelForSample] = dW_i[labelForSample] - currentDataPoint
    return dW_i

def main():
    x_train = np.array([[1, 5, 1, 4],
                        [2, 4, 0, 3],
                        [2, 1, 3, 3],
                        [2, 0, 4, 2],
                        [5, 1, 0, 2],
                        [4, 2, 1, 1]])
    y_train = [0, 0, 1, 1, 2, 2]
    x_test = np.array([[1, 5, 2, 4],
                       [2, 1, 2, 3],
                       [4, 1, 0, 1]])
    y_test = [0, 1, 2]
    W = np.array([[-1, 2,    1, 3],
                  [ 2, 0, -1, 4],
                  [ 1, 3,    2, 1]])

    delta = 1
    step_size = 0.01
     loss_L = 0
```

```
        dW = np.zeros(W.shape)
        prev_loss = 100
        '''# exercise 6 '''
        for idx, xsample in enumerate(x_train):
            s = predict(xsample,W)
            loss_i = computeLossForASample(s, y_train[idx], delta)
            print("Scores for sample {} with label {} is: {} and loss is {}".format(idx, y_train[idx], s,
          loss_i))
            dW_i = computeLossGradientForASample(W, s, x_train[idx], y_train[idx], delta)
            loss_L = loss_L + loss_i
            dW = dW + dW_i
        loss_L = loss_L    / len(x_train)
        print("The global normalized loss = {}".format(loss_L))
        dW = dW / len(x_train)
        W = W - step_size * dW
        print("The epoch is", 1, "The global normalized loss = {}".format(loss_L))

        '''# exercise 6 '''
        for i in range(1000):
            for idx, xsample in enumerate(x_train):
                s = predict(xsample,W)
                loss_i = computeLossForASample(s, y_train[idx], delta)
                print("Scores for sample {} with label {} is: {} and loss is {}".format(idx, y_train[idx], s,
loss_i))
                dW_i = computeLossGradientForASample(W, s, x_train[idx], y_train[idx], delta)
                loss_L = loss_L + loss_i
                dW = dW + dW_i
            #print(len(x_train))
            loss_L = loss_L    / len(x_train)
            print("The epoch is", i + 2, ",The global normalized loss = {}".format(loss_L))
            dW = dW / len(x_train)
            W = W - step_size * dW
            if loss_L < 0.001:
                break
        '''Exercise 7 - After solving exercise 6, predict the labels for the points existent in x_test variable'''
        correctPredicted = 0
        for idx, xsample in enumerate(x_test):
            predictedLabel = np.argmax(np.dot(W, xsample))
            if predictedLabel == y_test[idx]:
                correctPredicted = correctPredicted + 1
        accuracy = correctPredicted / len(x_test) * 100
        print("Accuracy for test = {}%".format(accuracy))
        return

    if __name__ == '__main__':
        main()
```

Comment: Define calls to the functions predict(), computeLossForASample() and computeLossGradientForASample() to calculate the gradient for each sample. A loop is done in exercise 6 to

determine the number of steps needed, update the weights, calculate the loss function, and then determine if the loss function value is less than 0.001 in an if conditional statement. A loop and an if conditional statement are done in exercise7 to obtain the accuracy.

Code for Exercise 8

```
import numpy as np
import pandas as pd
from sklearn.utils import shuffle
from sklearn.model_selection import train_test_split
from sklearn.model_selection import KFold

def predict(xsample, weights):
    s = []
    s = np.dot(weights, xsample)
    return s

def computeLossForASample(s, labelForSample, delta):
    loss_i = 0
    syi = s[labelForSample]
    for i in range(3):
        if i == labelForSample:
            continue
        loss_i = max(0, s[i] - syi + delta) + loss_i
    return loss_i

def computeLossGradientForASample(weights, s, currentDataPoint, labelForSample, delta):

    dW_i = np.zeros(weights.shape)
    syi = s[labelForSample]
    for j, sj in enumerate(s):
        dist = sj - syi + delta
        if j == labelForSample:
            continue
        if dist > 0:
            dW_i[j] = currentDataPoint
            dW_i[labelForSample] = dW_i[labelForSample] - currentDataPoint
    return dW_i

def main():
    dataset = pd.read_csv('Iris.csv', header = 1)    # list
    x = dataset.iloc[:, 0:4]
    y = dataset.iloc[:, 4]
    y = y.values
    x = x.values
    df = pd.DataFrame(np.array(y), columns=['class'])
    class_dict = df['class'].unique().tolist()
    df['label'] = df['class'].apply(lambda x_1: class_dict.index(x_1))
    y = df.iloc[:, 1]
```

```
        y = y.values
        x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.19, shuffle=True)
        #Question 1:    the optimal value for the weights adjustment step in order to obtain the maximum testing accuracy
        for step_size in [1, 0.5, 0.1, 0.05, 0.01, 0.005, 0.001, 0.0005, 0.0001, 0.00005, 0.00001]:
            loss_L = 0
            delta = 1
            AccMax = 0
            OptimalSteps = 0
            weights = np.array([[0.64256242,0.27115151,0.83237681, 0.17582961],
                                [0.24237855,0.77386685,0.38843292, 0.07962604],
                                [0.11121962,0.32523504,0.646248,      0.19051809]])
            dW = np.zeros(weights.shape)
            for i in range(10000):
                for idx, xsample in enumerate(x_train):
                    s = predict(xsample, weights)
                    loss_i = computeLossForASample(s, y_train[idx], delta)
                    print("Scores for sample {} with label {} is: {} and loss is {}".format(idx, y_train[idx], s, loss_i))
                    dW_i = computeLossGradientForASample(weights, s, x_train[idx], y_train[idx], delta)
                    loss_L = loss_L + loss_i
                    dW = dW + dW_i
                loss_L = loss_L / len(x_train)
                print("The epoch is",    i + 1, "The global normalized loss = {}".format(loss_L))
                dW = dW / len(x_train)
                weights = weights - step_size * dW

                correctPredicted = 0
                for idx, xsample in enumerate(x_test):
                    predictedLabel = np.argmax(np.dot(weights, xsample))
                    if predictedLabel == y_test[idx]:
                        correctPredicted = correctPredicted + 1
                accuracy = correctPredicted / len(x_test) * 100
                if accuracy > AccMax:
                    AccMax = accuracy
                    OptimalSteps = i + 1
            print("Learning Rate = {}".format(step_size), "The Maximum accuracy is {}% ,".format(AccMax),"The optimal step value is {}".format(OptimalSteps))
        return

    #Question 2:    What is the minimum number of steps necessary to train the system in order to obtain an accuracy superior to 90%?
    if accuracy > AccMax:
                    AccMax = accuracy
                    OptimalSteps = i + 1
                if accuracy > 90:
                    #print(accuracy)
                    NewOptimalSteps = i + 1
```

```
                    CurrentAccuracy = accuracy
                    print("Learning Rate = {}".format(step_size), "The Maximum accuracy is
{}% ,".format(CurrentAccuracy),
                          "The optimal step value is {}".format(NewOptimalSteps))
                    break

#Question 3: Is the system influenced by the random initialization of the weights matrix? Justify your
answer?

    if accuracy > AccMax:
                AccMax = accuracy
                NewOptimalSteps = i + 1
            if BestLoss > loss_L:
                BestLoss = loss_L
                # OptimalSteps = i + 1
        print("Learning Rate = {}".format(step_size), "The Maximum accuracy is
{}% ,".format(AccMax),
                  "The optimal step value is {}".format(NewOptimalSteps),"The lowest Loss in training
set = {}".format(BestLoss))
        plt.plot(Acc, alpha=0.8, label='Test Accuracy')
        plt.title("Test Accuracy Curve",    fontsize = 20)
        plt.xlabel("Training Steps")
        plt.savefig('./AccCurveRandom_1')

        plt.cla()
        plt.plot(loss, alpha = 0.8, label='Test Accuracy')
        plt.title("Training Loss Curve",    fontsize = 20)
        plt.xlabel("Training Steps")
        plt.savefig('./LossCurveRandom_1')

    if __name__ == '__main__':
        main()
```

Comment: The blue font represents the code for question one, the pink font represents the code for question two, and the purple font represents the code for question three. In problem 1, a fixed learning rate and randomly generated weights are set, looped in an epoch of 10000, and then an inline loop and two if conditional statements are applied in the loop to derive the maximum test accuracy and the optimal value of the weight adjustment step. In problem 2, the code from problem 1 is used to find the minimum steps required to train the system above 90% accuracy by comparing the current maximum accuracy with the original accuracy. In problem 3, the system is visualized by drawing a graph showing that the system is affected by the random initialization of the weight matrix.