Handwriting Digit Recognition

By: Yash Mathur

Problem 1 (KNN)

After downloading the data directly, download_mnist.py had to be changed to reflect my local directory changes. The file path for all data files in the 'filename' variable was preceded with 'mnist/' to properly reference the folder containing the data.

knn.py source code:

```
import math
import numpy as np
from download_mnist import load
import time
# classify using kNN
#x_train = np.load('../x_train.npy')
#y_train = np.load('../y_train.npy')
#x_test = np.load('../x_test.npy')
#y_test = np.load('../y_test.npy')
x_train, y_train, x_test, y_test = load()
x_train = x_train.reshape(60000,28,28)
x_test  = x_test.reshape(10000,28,28)
x_train = x_train.astype(float)
x_test = x_test.astype(float)

#L1 distance function
def d1(I1, I2):
   if I1.shape != I2.shape:
       raise ValueError("L1 Error: vectors must have the same dimensions.")
   sum = 0
   for i in range(I1.shape[0]):
       for j in range(I1.shape[1]):
           sum += abs(I1[i,j] - I2[i,j])

   return sum

#L2 distance function
def d2(I1, I2):
```

```
    if I1.shape != I2.shape:
        raise ValueError("L2 Error: vectors must have the same dimensions.")
    sum = 0
    for i in range(I1.shape[0]):
        for j in range(I1.shape[1]):
            sum += (I1[i,j] - I2[i,j])**2

    return math.sqrt(sum)

def kNNClassify(newInput, dataSet, labels, k):
    result=[]
    ########################
    # Input your code here #
    ########################
    for input in newInput:
        distances = []
        for i in range(dataSet.shape[0]):
            #set distance metric here
            distance = d1(input, dataSet[i])
            distances.append(distance)

        #returns k smallest distance indices
        idxs = np.argpartition(distances, k)
        count = [0] * 10

        for idx in idxs[:k]:
            count[labels[idx]] += 1

        result.append(np.argmax(count))



    ####################
    # End of your code #
    ####################
    return result

accuracies = []
for k in range(25):
    start_time = time.time()
    outputlabels=kNNClassify(x_test[0:25],x_train[:2500],y_train[:2500], k)
    result = y_test[0:25] - outputlabels
```

```
    result = (1 - np.count_nonzero(result)/len(outputlabels))
    accuracies.append(result)
    print("---utilizing %s nearest neighbors ---" %k)
    print ("---classification accuracy for knn on mnist: %s ---" %result)
    print ("---execution time: %s seconds ---" % (time.time() - start_time))
print("Highest accuracy achieved with %s nearest neighbors" %np.argmax(accuracies))
```

KNN epochs were run providing 2500 samples of training data with labels and 25 test samples to determine the accuracy of the KNN classifier for 25 values of k (0:24). The KNN was tested using both the L1 and L2 distance metrics. Hence, in total, 50 epochs of the KNN classifier ran. The outputs of the KNN utilizing both distance metrics are recorded below.

KNN output using L1 (Manhattan) distance metric:
---utilizing 0 nearest neighbors ---
---classification accuracy for knn on mnist: 0.12 ---
---execution time: 23.49489974975586 seconds ---
---utilizing 1 nearest neighbors ---
---classification accuracy for knn on mnist: 0.8 ---
---execution time: 23.2182400226593 seconds ---
---utilizing 2 nearest neighbors ---
---classification accuracy for knn on mnist: 0.92 ---
---execution time: 23.415313005447388 seconds ---
---utilizing 3 nearest neighbors ---
---classification accuracy for knn on mnist: 0.96 ---
---execution time: 23.298596143722534 seconds ---
---utilizing 4 nearest neighbors ---
---classification accuracy for knn on mnist: 0.92 ---
---execution time: 22.2171630859375 seconds ---
---utilizing 5 nearest neighbors ---
---classification accuracy for knn on mnist: 0.92 ---
---execution time: 23.98344874382019 seconds ---
---utilizing 6 nearest neighbors ---
---classification accuracy for knn on mnist: 0.88 ---
---execution time: 22.240367889404297 seconds ---
---utilizing 7 nearest neighbors ---
---classification accuracy for knn on mnist: 0.96 ---
---execution time: 23.358529806137085 seconds ---
---utilizing 8 nearest neighbors ---
---classification accuracy for knn on mnist: 0.96 ---
---execution time: 22.782378911972046 seconds ---
---utilizing 9 nearest neighbors ---

---classification accuracy for knn on mnist: 0.96 ---
---execution time: 23.611552000045776 seconds ---
---utilizing 10 nearest neighbors ---
---classification accuracy for knn on mnist: 0.96 ---
---execution time: 22.519452810287476 seconds ---
---utilizing 11 nearest neighbors ---
---classification accuracy for knn on mnist: 0.96 ---
---execution time: 22.595486879348755 seconds ---
---utilizing 12 nearest neighbors ---
---classification accuracy for knn on mnist: 0.92 ---
---execution time: 22.31774115562439 seconds ---
---utilizing 13 nearest neighbors ---
---classification accuracy for knn on mnist: 0.96 ---
---execution time: 22.74449920654297 seconds ---
---utilizing 14 nearest neighbors ---
---classification accuracy for knn on mnist: 0.92 ---
---execution time: 22.865463972091675 seconds ---
---utilizing 15 nearest neighbors ---
---classification accuracy for knn on mnist: 0.92 ---
---execution time: 22.236870050430298 seconds ---
---utilizing 16 nearest neighbors ---
---classification accuracy for knn on mnist: 0.96 ---
---execution time: 22.1914119720459 seconds ---
---utilizing 17 nearest neighbors ---
---classification accuracy for knn on mnist: 0.92 ---
---execution time: 22.092159032821655 seconds ---
---utilizing 18 nearest neighbors ---
---classification accuracy for knn on mnist: 0.96 ---
---execution time: 21.860617876052856 seconds ---
---utilizing 19 nearest neighbors ---
---classification accuracy for knn on mnist: 0.96 ---
---execution time: 22.385531902313232 seconds ---
---utilizing 20 nearest neighbors ---
---classification accuracy for knn on mnist: 0.96 ---
---execution time: 22.125306844711304 seconds ---
---utilizing 21 nearest neighbors ---
---classification accuracy for knn on mnist: 0.96 ---
---execution time: 22.713068962097168 seconds ---
---utilizing 22 nearest neighbors ---
---classification accuracy for knn on mnist: 0.96 ---

---execution time: 22.379616022109985 seconds ---
---utilizing 23 nearest neighbors ---
---classification accuracy for knn on mnist: 0.96 ---
---execution time: 22.49427890777588 seconds ---
---utilizing 24 nearest neighbors ---
---classification accuracy for knn on mnist: 0.96 ---
---execution time: 20.767493963241577 seconds ---
Highest accuracy achieved with 3 nearest neighbors

KNN output using L2 (Euclidean) distance metric:
---utilizing 0 nearest neighbors ---
---classification accuracy for knn on mnist: 0.12 ---
---execution time: 23.6735680103302 seconds ---
---utilizing 1 nearest neighbors ---
---classification accuracy for knn on mnist: 0.8 ---
---execution time: 22.687017917633057 seconds ---
---utilizing 2 nearest neighbors ---
---classification accuracy for knn on mnist: 0.88 ---
---execution time: 24.40127992630005 seconds ---
---utilizing 3 nearest neighbors ---
---classification accuracy for knn on mnist: 0.96 ---
---execution time: 24.06628394126892 seconds ---
---utilizing 4 nearest neighbors ---
---classification accuracy for knn on mnist: 0.92 ---
---execution time: 24.4288330078125 seconds ---
---utilizing 5 nearest neighbors ---
---classification accuracy for knn on mnist: 0.96 ---
---execution time: 24.27123785018921 seconds ---
---utilizing 6 nearest neighbors ---
---classification accuracy for knn on mnist: 0.92 ---
---execution time: 23.34577465057373 seconds ---
---utilizing 7 nearest neighbors ---
---classification accuracy for knn on mnist: 0.96 ---
---execution time: 24.6924569606781 seconds ---
---utilizing 8 nearest neighbors ---
---classification accuracy for knn on mnist: 0.96 ---
---execution time: 23.520236015319824 seconds ---
---utilizing 9 nearest neighbors ---
---classification accuracy for knn on mnist: 0.96 ---
---execution time: 24.203866958618164 seconds ---

---utilizing 10 nearest neighbors ---
---classification accuracy for knn on mnist: 0.96 ---
---execution time: 23.218929767608643 seconds ---
---utilizing 11 nearest neighbors ---
---classification accuracy for knn on mnist: 0.96 ---
---execution time: 24.55798578262329 seconds ---
---utilizing 12 nearest neighbors ---
---classification accuracy for knn on mnist: 0.96 ---
---execution time: 23.384930849075317 seconds ---
---utilizing 13 nearest neighbors ---
---classification accuracy for knn on mnist: 0.96 ---
---execution time: 23.51587414741516 seconds ---
---utilizing 14 nearest neighbors ---
---classification accuracy for knn on mnist: 0.96 ---
---execution time: 23.380799055099487 seconds ---
---utilizing 15 nearest neighbors ---
---classification accuracy for knn on mnist: 0.96 ---
---execution time: 23.693092823028564 seconds ---
---utilizing 16 nearest neighbors ---
---classification accuracy for knn on mnist: 0.96 ---
---execution time: 23.211714029312134 seconds ---
---utilizing 17 nearest neighbors ---
---classification accuracy for knn on mnist: 0.96 ---
---execution time: 23.112045764923096 seconds ---
---utilizing 18 nearest neighbors ---
---classification accuracy for knn on mnist: 0.96 ---
---execution time: 22.97667407989502 seconds ---
---utilizing 19 nearest neighbors ---
---classification accuracy for knn on mnist: 0.96 ---
---execution time: 22.795326948165894 seconds ---
---utilizing 20 nearest neighbors ---
---classification accuracy for knn on mnist: 0.96 ---
---execution time: 23.261754751205444 seconds ---
---utilizing 21 nearest neighbors ---
---classification accuracy for knn on mnist: 0.96 ---
---execution time: 22.984575033187866 seconds ---
---utilizing 22 nearest neighbors ---
---classification accuracy for knn on mnist: 0.96 ---
---execution time: 23.30801510810852 seconds ---
---utilizing 23 nearest neighbors ---

```
---classification accuracy for knn on mnist: 0.96 ---
---execution time: 23.52876305580139 seconds ---
---utilizing 24 nearest neighbors ---
---classification accuracy for knn on mnist: 0.96 ---
---execution time: 23.382943868637085 seconds ---
Highest accuracy achieved with 3 nearest neighbors
```

By observing the output, in general, odd k values result in higher accuracies. This makes sense because an odd number of neighbors ensure there is one clear predictive label when choosing the maximum from the frequencies of each neighbor's label. In addition, increasing the number of neighbors utilized in prediction doesn't necessarily improve the accuracy nor substantially increase the execution time of the model. Both distance metric algorithms resulted in the highest accuracy when using 3 nearest neighbors. The validity of the tests run on the KNN model can also be observed in the output, since utilizing 0 neighbors in the algorithm resulted in an accuracy of 12% roughly equivalent to the probability of guessing each digit.

Problem 16 (Linear Classifier)

linear_classifier.py source code:

```
import math
import numpy as np
from download_mnist import load
import time
# classify using kNN
#x_train = np.load('../x_train.npy')
#y_train = np.load('../y_train.npy')
#x_test = np.load('../x_test.npy')
#y_test = np.load('../y_test.npy')
x_train, y_train, x_test, y_test = load()
x_train = x_train.reshape(60000,28,28)
x_test  = x_test.reshape(10000,28,28)
x_train = x_train.astype(float)
x_test = x_test.astype(float)

def compress(img):
   v = []
   width = img.shape[0]
   height = img.shape[1]
   for i in range(width):
       for j in range(height):
           v.append(img[i][j])
```

```
    #append bias placeholder
    v.append(1)
    return np.array(v)

def cross_entropy_loss(y_true, y_pred):
    y_pred = np.exp(y_pred) / np.sum(np.exp(y_pred))
    loss = np.dot(y_true, np.log(y_pred))
    return loss

def train(x_train, y_train, epochs):
    best_loss = float('inf')
    best_W = None
    for epoch in range(epochs):
        W = np.random.randn(10, (28*28 + 1)) * 0.0001
        loss = 0
        for i, x in enumerate(x_train):
            x = compress(x)
            y = np.zeros(10)
            y[y_train[i]] = 1
            pred = np.dot(W, x)
            sample_loss = cross_entropy_loss(y, pred)
            loss += sample_loss
        loss /= x_train.shape[0]
        loss *= -1
        if loss < best_loss:
            best_loss = loss
            best_W = W
        print('In epoch %d the loss was %.4f, best %.4f' %(epoch, loss, best_loss))

    return best_W

def predict(x_test, W):
    result = []
    for x in x_test:
        x = compress(x)
        pred = np.dot(W, x)
        result.append(np.argmax(pred))
    return result

start_time = time.time()
W = train(x_train[:2500], y_train[:2500], 100)
```

```
outputlabels = predict(x_test[:25], W)
result = y_test[0:25] - outputlabels
result = (1 - np.count_nonzero(result)/len(outputlabels))
print ("---classification accuracy for linear classifier on mnist: %s ---" %result)
print ("---execution time: %s seconds ---" % (time.time() - start_time))
```

Using Random Search, 100 epochs of the linear classifier were run to determine the best Weight matrix. In each epoch the Cross Entropy Loss function was used to compare Weight matrices across epochs. In each epoch, 2500 training samples with labels were used. The final accuracy was determined using 25 test samples. The output of the linear classifier is recorded below.

Linear Classifier output:

In epoch 0 the loss was 2.2996, best 2.2996
In epoch 1 the loss was 2.3489, best 2.2996
In epoch 2 the loss was 2.3305, best 2.2996
In epoch 3 the loss was 2.3248, best 2.2996
In epoch 4 the loss was 2.3603, best 2.2996
In epoch 5 the loss was 2.3252, best 2.2996
In epoch 6 the loss was 2.3673, best 2.2996
In epoch 7 the loss was 2.3225, best 2.2996
In epoch 8 the loss was 2.3273, best 2.2996
In epoch 9 the loss was 2.2442, best 2.2442
In epoch 10 the loss was 2.3205, best 2.2442
In epoch 11 the loss was 2.3337, best 2.2442
In epoch 12 the loss was 2.3028, best 2.2442
In epoch 13 the loss was 2.3177, best 2.2442
In epoch 14 the loss was 2.3626, best 2.2442
In epoch 15 the loss was 2.3263, best 2.2442
In epoch 16 the loss was 2.3302, best 2.2442
In epoch 17 the loss was 2.3764, best 2.2442
In epoch 18 the loss was 2.3126, best 2.2442
In epoch 19 the loss was 2.3444, best 2.2442
In epoch 20 the loss was 2.3342, best 2.2442
In epoch 21 the loss was 2.3648, best 2.2442
In epoch 22 the loss was 2.3320, best 2.2442
In epoch 23 the loss was 2.3622, best 2.2442
In epoch 24 the loss was 2.2876, best 2.2442
In epoch 25 the loss was 2.2513, best 2.2442
In epoch 26 the loss was 2.2941, best 2.2442
In epoch 27 the loss was 2.3397, best 2.2442
In epoch 28 the loss was 2.2861, best 2.2442

In epoch 29 the loss was 2.3328, best 2.2442
In epoch 30 the loss was 2.3668, best 2.2442
In epoch 31 the loss was 2.2843, best 2.2442
In epoch 32 the loss was 2.3380, best 2.2442
In epoch 33 the loss was 2.2777, best 2.2442
In epoch 34 the loss was 2.3707, best 2.2442
In epoch 35 the loss was 2.3440, best 2.2442
In epoch 36 the loss was 2.3068, best 2.2442
In epoch 37 the loss was 2.3156, best 2.2442
In epoch 38 the loss was 2.2657, best 2.2442
In epoch 39 the loss was 2.3386, best 2.2442
In epoch 40 the loss was 2.2846, best 2.2442
In epoch 41 the loss was 2.3008, best 2.2442
In epoch 42 the loss was 2.2890, best 2.2442
In epoch 43 the loss was 2.3088, best 2.2442
In epoch 44 the loss was 2.2858, best 2.2442
In epoch 45 the loss was 2.3521, best 2.2442
In epoch 46 the loss was 2.3052, best 2.2442
In epoch 47 the loss was 2.2729, best 2.2442
In epoch 48 the loss was 2.3086, best 2.2442
In epoch 49 the loss was 2.3661, best 2.2442
In epoch 50 the loss was 2.3449, best 2.2442
In epoch 51 the loss was 2.3161, best 2.2442
In epoch 52 the loss was 2.3455, best 2.2442
In epoch 53 the loss was 2.2925, best 2.2442
In epoch 54 the loss was 2.3051, best 2.2442
In epoch 55 the loss was 2.3383, best 2.2442
In epoch 56 the loss was 2.3330, best 2.2442
In epoch 57 the loss was 2.3149, best 2.2442
In epoch 58 the loss was 2.3654, best 2.2442
In epoch 59 the loss was 2.3855, best 2.2442
In epoch 60 the loss was 2.3332, best 2.2442
In epoch 61 the loss was 2.3405, best 2.2442
In epoch 62 the loss was 2.3125, best 2.2442
In epoch 63 the loss was 2.2994, best 2.2442
In epoch 64 the loss was 2.4030, best 2.2442
In epoch 65 the loss was 2.3877, best 2.2442
In epoch 66 the loss was 2.3191, best 2.2442
In epoch 67 the loss was 2.3316, best 2.2442
In epoch 68 the loss was 2.3190, best 2.2442

In epoch 69 the loss was 2.2590, best 2.2442
In epoch 70 the loss was 2.3135, best 2.2442
In epoch 71 the loss was 2.3505, best 2.2442
In epoch 72 the loss was 2.3487, best 2.2442
In epoch 73 the loss was 2.3490, best 2.2442
In epoch 74 the loss was 2.3657, best 2.2442
In epoch 75 the loss was 2.3826, best 2.2442
In epoch 76 the loss was 2.3539, best 2.2442
In epoch 77 the loss was 2.3222, best 2.2442
In epoch 78 the loss was 2.3008, best 2.2442
In epoch 79 the loss was 2.3711, best 2.2442
In epoch 80 the loss was 2.3353, best 2.2442
In epoch 81 the loss was 2.3385, best 2.2442
In epoch 82 the loss was 2.3340, best 2.2442
In epoch 83 the loss was 2.3052, best 2.2442
In epoch 84 the loss was 2.3393, best 2.2442
In epoch 85 the loss was 2.3501, best 2.2442
In epoch 86 the loss was 2.3445, best 2.2442
In epoch 87 the loss was 2.3513, best 2.2442
In epoch 88 the loss was 2.3498, best 2.2442
In epoch 89 the loss was 2.3342, best 2.2442
In epoch 90 the loss was 2.3240, best 2.2442
In epoch 91 the loss was 2.3381, best 2.2442
In epoch 92 the loss was 2.3055, best 2.2442
In epoch 93 the loss was 2.2813, best 2.2442
In epoch 94 the loss was 2.3428, best 2.2442
In epoch 95 the loss was 2.3365, best 2.2442
In epoch 96 the loss was 2.3811, best 2.2442
In epoch 97 the loss was 2.3522, best 2.2442
In epoch 98 the loss was 2.3365, best 2.2442
In epoch 99 the loss was 2.3412, best 2.2442
---classification accuracy for linear classifier on mnist: 0.36 ---
---execution time: 56.708898067474365 seconds —