# Nityash_Gautam_Assignment_2

May 6, 2023

## 1 This exercise will focus on training a neural network classifier for the MNIST dataset.


1. NAME: NITYASH GAUTAM
2. SID: 862395403
3. UCR MAIL ID: ngaut006@ucr.edu


### 1.1 Importing Essentials

[1]:
```
from keras.datasets import mnist
from keras.utils import np_utils
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import accuracy_score
import math
from sklearn.utils import shuffle
from numpy.ma.extras import unique
import time
```

### 1.2 Main Assignment Tasks Begin

#### 1.2.1 TASK 1: (2 pts)

Apply Normalization on Training and Test Data

[2]:
```
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-
datasets/mnist.npz
11490434/11490434 [==============================] - 0s 0us/step
```

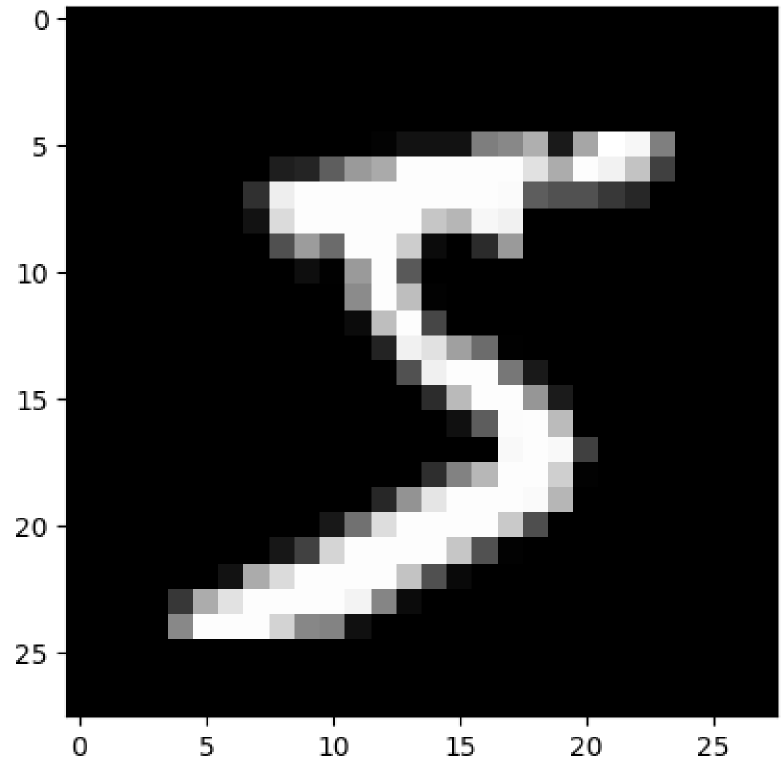
[3]:
```
print(x_train.shape, y_train.shape, x_test.shape, y_test.shape)
```

```
(60000, 28, 28) (60000,) (10000, 28, 28) (10000,)
```

[4]:
```
# Visualize 1 sample
print ('label', y_train[0])
plt.imshow(x_train[0], cmap='gray')
```

label 5

[4]: <matplotlib.image.AxesImage at 0x7f7e2d810f70>




**Input: Each input is a 28 x 28 matri. Apply the following operations to obtain d = 785 dimensional input features**

**Convert Inputs x to vectors of size $28^2$ = 784**

[5]:
```
x_train = x_train.reshape(x_train.shape[0], -1)
x_test = x_test.reshape(x_test.shape[0], -1)
```

[6]:
```
print("Train Data Shape = ", x_train.shape)
print("Test Data Shape = ", x_test.shape)
```

```
Train Data Shape =  (60000, 784)
Test Data Shape =  (10000, 784)
```

**Standardize input images using *z-normalization*: Scale each image *i* ($1 \leq i \leq N$) to have zero mean and unit variance. In Python Terms, this corresponds to the operation** $x => \frac{x - np.mean(x)}{np.std(x)}$.

[7]:
```
x_train = (x_train - np.mean(x_train, axis=1, keepdims=True)) / (np.
 ↪std(x_train, axis=1, keepdims=True) + 1e-8)
x_test = (x_test - np.mean(x_test, axis=1, keepdims=True)) / (np.std(x_test,␣
 ↪axis=1, keepdims=True) + 1e-8)
```

[8]:
```
print("Shape of Normalized Train Data = ", x_train.shape)
print("Shape of Normalized Test Data = ", x_test.shape)
```

```
Shape of Normalized Train Data =  (60000, 784)
Shape of Normalized Test Data =  (10000, 784)
```

**Add bias variable by concatenating 1 to your input. Making the input dimension becomes d = 785.**

[9]:
```
x_train = np.hstack([x_train, np.ones((x_train.shape[0], 1))])
x_test = np.hstack([x_test, np.ones((x_test.shape[0], 1))])
```

[10]:
```
print("Shape of Normalized train data after appending 1 = ", x_train.shape)
print("Shape of Normalized test data after appending 1 = ", x_test.shape)
```

```
Shape of Normalized train data after appending 1 =  (60000, 785)
Shape of Normalized test data after appending 1 =  (10000, 785)
```

**Output: Each label $y$ is a digit from 0 to 9. Convert $y$ to 0,1 as follows:**

$$y->[\frac{0-0\leq y\leq 4}{1-5\leq y\leq 9}$$

[11]:
```
y_train = (y_train > 4).astype('int')
y_test = (y_test > 4).astype('int')
```

[12]:
```
print(y_train.shape)
print(y_test)
```

```
(60000,)
[1 0 0 … 0 1 1]
```

**Final Checks on Data**

[13]:
```
# Check if the data contains NaN or infinite values
print("Train data contains NaN:", np.any(np.isnan(x_train)))
print("Train data contains infinite values:", np.any(np.isinf(x_train)))
print()
print("Test data contains NaN:", np.any(np.isnan(x_test)))
print("Test data contains infinite values:", np.any(np.isinf(x_test)))
print()

# Check if the labels are correctly assigned
print("Unique train labels:", np.unique(y_train))
```

```python
print("Unique test labels:", np.unique(y_test))
print()

# Check the shapes of the data and labels
print("Train data shape:", x_train.shape)
print("Train labels shape:", y_train.shape)
print()
print("Test data shape:", x_test.shape)
print("Test labels shape:", y_test.shape)
```

```
Train data contains NaN: False
Train data contains infinite values: False

Test data contains NaN: False
Test data contains infinite values: False

Unique train labels: [0 1]
Unique test labels: [0 1]

Train data shape: (60000, 785)
Train labels shape: (60000,)

Test data shape: (10000, 785)
Test labels shape: (10000,)
```

### 1.2.2 TASK 2: (2 pts)

As a baseline, train a linear classifier $y = v^T x$ and quadratic loss. Report its test accuracy

**Helper Functions**

[14]:
```python
def convert_to_one_hot(labels):
    """
    Convert an array of labels into a one-hot encoded array.

    Input:
      ndarray): Array of labels to be converted to one-hot encoding.

    Returns:
      numpy.ndarray: One-hot encoded array of labels.
    """
    unique = np.unique(labels) # Get the unique labels
    onehot = np.zeros((labels.shape[0], unique.shape[0])) # Create an array of␣
 ↪zeros with dimensions same as labels
    onehot[np.arange(labels.shape[0]), labels] = 1 # Set the corresponding␣
 ↪position of each label in onehot array to 1
    return onehot # Return the one-hot encoded array
```

[15]:
```python
def accuracy(y_true, onehot_y_out):
    """
    Calculate the accuracy of a classifier's predictions.

    Input:
      y_true (numpy.ndarray): Array of true labels.
      onehot_y_out (numpy.ndarray): One-hot encoded array of predicted labels.

    Returns:
      float: Accuracy of the classifier's predictions.
    """
    predicted_labels = np.argmax(onehot_y_out, axis=1) # Get the predicted␣
↪labels by finding the index of the max value in each row
    correct_predictions = np.sum(predicted_labels == y_true) # Get the number␣
↪of correct predictions
    accuracy = correct_predictions / y_true.shape[0] # Calculate the accuracy␣
↪by dividing the number of correct predictions by the total number of␣
↪predictions
    return accuracy # Return the accuracy as a float
```

[16]:
```python
def predict(X, w):
    """
    Make predictions based on the input and weights.

    Input:
      X (numpy.ndarray): Array of input data.
      w (numpy.ndarray): Array of weights.

    Returns:
      numpy.ndarray: Array of predictions.
    """
    predictions = np.matmul(X, w.T) # Calculate the dot product of the input␣
↪data and the weights transposed
    return predictions # Return the array of predictions
```

[17]:
```python
def loss(onehot_y_pred, onehot_y_true):
    """
    Calculate the mean squared error (MSE) loss between predicted and true␣
↪labels.

    Input:
      onehot_y_pred (numpy.ndarray): One-hot encoded array of predicted labels.
      onehot_y_true (numpy.ndarray): One-hot encoded array of true labels.

    Returns:
      float: Mean squared error (MSE) loss between predicted and true labels.
    """
```

```
    diff = onehot_y_pred - onehot_y_true # Calculate the difference between the␣
↪predicted and true labels
    squared_diff = diff**2 # Square the difference
    mean_squared_diff = np.sum(squared_diff)/(2*onehot_y_pred.shape[0]) #␣
↪Calculate the mean squared difference
    return mean_squared_diff # Return the mean squared error loss as a float
```

[18]:
```
def plot_accuracy(train_accuracy, test_accuracy):
    plt.figure(2)
    plt.plot(np.arange(0,len(train_accuracy)), train_accuracy, label='Train␣
↪Accuracy')
    plt.plot(np.arange(0,len(test_accuracy)), test_accuracy, label='Test␣
↪Accuracy')
    plt.title('Plot of ACCURACY vs ITERATIONS')
    plt.xlabel('ITERATIONS')
    plt.ylabel('ACCURACY')
    plt.legend()
    plt.grid()
    plt.show()
```

**Main Function**

[19]:
```
def linear_model(x_train, y_train_true, y_train_oh, x_test, y_test_true,␣
↪y_test_oh, lr=0.001, n_epochs=50, batch_size=10):
    """
    Train a linear model on the training set and evaluate it on the test set.

    Input:
      x_train (numpy.ndarray): Array of training data.
      y_train_true (numpy.ndarray): Array of true training labels.
      y_train_oh (numpy.ndarray): One-hot encoded array of training labels.
      x_test (numpy.ndarray): Array of test data.
      y_test_true (numpy.ndarray): Array of true test labels.
      y_test_oh (numpy.ndarray): One-hot encoded array of test labels.
      lr (float): Learning rate (default=0.001).
      n_epochs (int): Number of training epochs (default=50).
      batch_size (int): Batch size for training (default=10).

    Returns:
      - numpy.ndarray: Array of weights learned during training.
      - numpy.ndarray: Array of training losses.
      - numpy.ndarray: Array of training accuracies.
      - numpy.ndarray: Array of test accuracies.
    """
    # Initialize weights
    input_dim_linear = x_train.shape[1]
    output_dim_linear = y_train_oh.shape[1]
```

```
    weights_linear = np.zeros((output_dim_linear, input_dim_linear))

    # Initialize arrays to store training progress
    loss_array_linear = []
    train_accuracy_array_linear = []
    test_accuracy_array_linear = []

    for epoch in range(n_epochs):
        # Shuffle data for each epoch
        shuff_idx = np.random.permutation(x_train.shape[0])
        x_shuffled_linear = x_train[shuff_idx]
        onehot_y_shuffled_linear = y_train_oh[shuff_idx]

        # Update weights in batches
        i = 0
        while i < x_train.shape[0]:
            x = x_shuffled_linear[i:i + batch_size]
            y = onehot_y_shuffled_linear[i:i + batch_size]

            out = predict(x, weights_linear)
            l_linear = loss(out, y)
            w_grad = np.matmul((out - y).T, x) / out.shape[0]
            weights_linear -= lr * w_grad

            i += batch_size

        # Calculate training and test accuracy for this epoch
        loss_array_linear.append(l_linear)
        train_acc = accuracy(y_train_true, predict(x_train, weights_linear))
        train_accuracy_array_linear.append(train_acc)
        test_acc = accuracy(y_test_true, predict(x_test, weights_linear))
        test_accuracy_array_linear.append(test_acc)

        # Print progress every 10 epochs
        if (epoch + 1) % 10 == 0:
            print(f'epoch = {epoch+1}, Training Loss = {l_linear}, Training␣
↪Accuracy = {train_acc}, Test Accuracy = {test_acc}')

    return weights_linear, loss_array_linear, train_accuracy_array_linear,␣
↪test_accuracy_array_linear
```

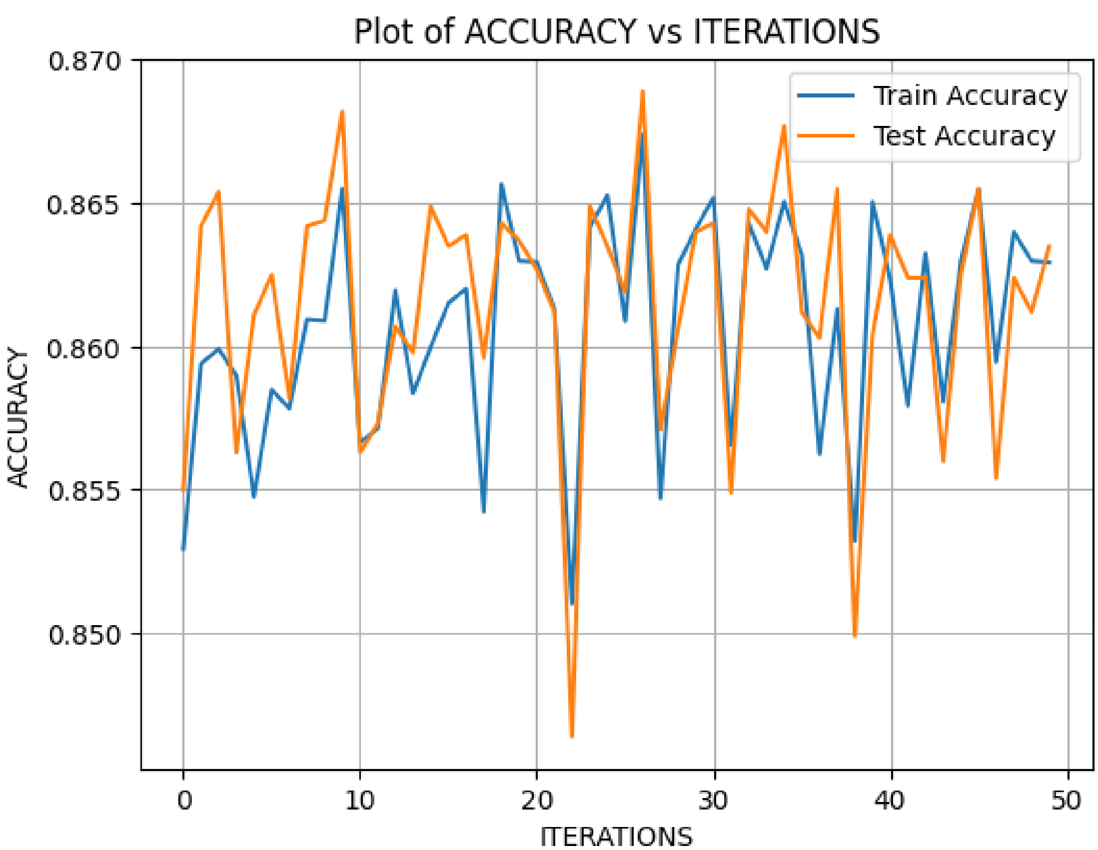
[20]:
```
# convert y to one hot encoded
onehot_y_train_linear = convert_to_one_hot(y_train)
onehot_y_test_linear = convert_to_one_hot(y_test)
```

**Testing The Linear Classifier**

[21]:
```
w,loss_arr,train_acc, test_acc = linear_model(x_train, y_train,␣
↪onehot_y_train_linear,x_test, y_test, onehot_y_test_linear)
```

```
epoch = 10, Training Loss = 0.10064246998312809, Training Accuracy = 0.8655,
Test Accuracy = 0.8682
epoch = 20, Training Loss = 0.17115160514195743, Training Accuracy =
0.8629833333333333, Test Accuracy = 0.8637
epoch = 30, Training Loss = 0.06669519074421805, Training Accuracy = 0.8641,
Test Accuracy = 0.864
epoch = 40, Training Loss = 0.09203983775927589, Training Accuracy = 0.86505,
Test Accuracy = 0.8604
epoch = 50, Training Loss = 0.20810575292426203, Training Accuracy =
0.8629333333333333, Test Accuracy = 0.8635
```

[22]:
```
plot_accuracy(train_acc, test_acc)
```




[23]:
```
print ("The test accuracy is {}%".format(test_acc[-1] * 100))
```

```
The test accuracy is 86.35000000000001%
```

### 1.2.3 TASK 3: (7 pts)

Train a neural network classifier with quadratic loss (y, f(x)) = (y − f(x))2. Plot the progress of the test and training accuracy (y-axis) as a function of the iteration counter t (x-axis)2. Report the final test accuracy for the following choices: * k = 5 * k = 40 * k = 200 * Comment on the role of hidden units k on the ease of optimization and accuracy.

**Helper Functions**

[24]:
```
def quadratic_loss(y_true, y_pred):
    """
    Calculate the mean quadratic loss between predicted and true values.

    Input:
      y_true (numpy.ndarray): Array of true values.
      y_pred (numpy.ndarray): Array of predicted values.

    Returns:
      float: The mean quadratic loss between predicted and true values.
    """
    return np.mean((y_true - y_pred)**2)
```

[25]:
```
def relu(x):
    """
    Rectified Linear Unit (ReLU) activation function.

    Input:
      input (numpy.ndarray): Array of input values.

    Returns:
      numpy.ndarray: Array of values resulting from applying ReLU.
    """
    return np.maximum(x, 0)
```

[26]:
```
def relu_deriv(x):
    """
    Derivative of the Rectified Linear Unit (ReLU) activation function.

    Input:
      input (numpy.ndarray): Array of input values.

    Returns:
      numpy.ndarray: Array of values resulting from applying the ReLU␣
 ↪derivative.
    """
    return np.where(x > 0, 1, 0)
```

```
[27]: def get_accuracy_ql(true_y, pred_y):
    """
    Calculate accuracy of predicted values using quadratic loss.

    Input:
      y_true (numpy.ndarray): Array of true values.
      y_pred (numpy.ndarray): Array of predicted values.

    Returns:
      float: The accuracy of predicted values using quadratic loss.
    """
    pred_y = pred_y.reshape(-1,)
    pred_y = np.where(pred_y > 0, 1, 0)

    return np.sum(true_y == pred_y)/true_y.shape[0]
```

**Main Function**

```
[28]: def shallow_neural_quadratic(x_train, x_test, y_train, y_test, lr=0.01, k=5,␣
 ↪epochs=10, batch_size=10):
    """
    This function trains a shallow neural network with a quadratic layer and␣
 ↪calculates training and testing accuracy.

    Input:
      x_train (numpy array): Training input data.
      x_test (numpy array): Testing input data.
      y_train (numpy array): Training output data.
      y_test (numpy array): Testing output data.
      lr (float, optional): Learning rate. Default is 0.01.
      k (int, optional): Number of hidden units. Default is 5.
      epochs (int, optional): Number of training epochs. Default is 10.
      batch_size (int, optional): Size of the mini-batch. Default is 10.

    Returns:
      A tuple containing two lists (train_acc_per_iteration_qd,␣
 ↪test_acc_per_iteration_qd) with the training and testing accuracy per␣
 ↪iteration.
    """

    # Set random seed for reproducibility
    np.random.seed(112233)

    # Initialize weights for the quadratic layer
    w_qd = np.random.randn(x_train.shape[1], k) / np.sqrt(x_train.shape[1])
    v_qd = np.random.randn(k) / np.sqrt(k)
```

```
# Initialize lists to store training and testing accuracy per iteration
train_acc_per_iteration_qd = []
test_acc_per_iteration_qd = []

# Initialize iteration counter
iter_ctr_qd = 0

# Start training loop
for epoch in range(epochs):

    # Shuffle the training data
    shuffled_indices = np.random.permutation(x_train.shape[0])
    x_shuffled = x_train[shuffled_indices]
    y_shuffled = y_train[shuffled_indices]

    i = 0

    # Process the data in mini-batches
    while i < x_train.shape[0]:

        # Get the current mini-batch
        x = x_shuffled[i:i+batch_size]
        y = y_shuffled[i:i+batch_size]

        # Perform forward pass
        z1 = np.matmul(x, w_qd)
        y1 = relu(z1)
        z2 = np.matmul(y1, v_qd)
        y2 = np.round(z2)

        # Calculate error
        delta_2 = 2*(y2 - y)

        # Calculate weight update for output layer
        dv = np.matmul(y1.T, delta_2) / x.shape[0]

        # Calculate weight update for hidden layer
        relu_derivative = relu_deriv(z1)
        delta_1 = np.matmul(delta_2.reshape(-1,1), v_qd.reshape(-1,1).T) *
↪relu_derivative
        dw = np.matmul(x.T, delta_1) / x.shape[0]

        # Update weights
        w_qd -= lr*dw
        v_qd -= lr*dv

        # Increment mini-batch counter
```

```
            i += batch_size

            # Calculate training and testing accuracy every 10000 iterations or␣
 ↪at the start
            if iter_ctr_qd == 0 or iter_ctr_qd % 10000 == 0:
                z1 = np.matmul(x_train, w_qd)
                y1 = relu(z1)
                z2 = np.matmul(y1, v_qd)
                y2 = np.round(z2)
                train_accuracy_ql = get_accuracy_ql(y_train, y2)
                z1 = np.matmul(x_test, w_qd)
                y1 = relu(z1)
                z2 = np.matmul(y1, v_qd)
                y2 = np.round(z2)
                test_accuracy_ql = get_accuracy_ql(y_test, y2)

                # Append training and testing accuracy to their respective lists
                train_acc_per_iteration_qd.append((iter_ctr_qd,␣
 ↪train_accuracy_ql))
                test_acc_per_iteration_qd.append((iter_ctr_qd,␣
 ↪test_accuracy_ql))

            # Increment iteration counter
            iter_ctr_qd += 1

        # Print training and testing accuracy for the current epoch
        print('For the EPOCH:', epoch + 1, ' Training Accuracy =',␣
 ↪train_accuracy_ql, ' Testing Accuracy =', test_accuracy_ql)

    # Return training and testing accuracy per iteration
    return train_acc_per_iteration_qd, test_acc_per_iteration_qd
```

### Testing the Shallow Net with Quadratic Loss

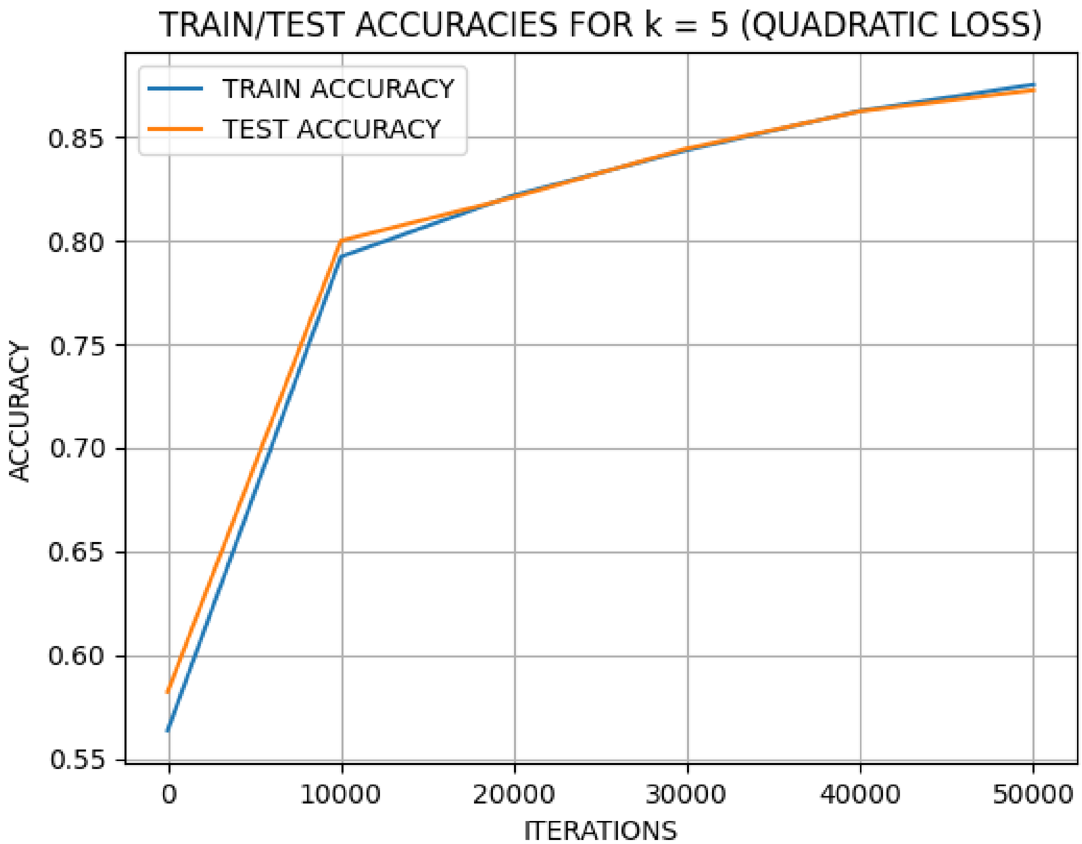
For k = 5

[29]:
```
print('For K = 5')

print()

# Call the function to train a shallow neural network with quadratic loss on␣
 ↪the input data
train_acc_per_iteration_ql, test_acc_per_iteration_ql =␣
 ↪shallow_neural_quadratic(x_train, x_test, y_train, y_test, lr=0.001, k=5,␣
 ↪epochs=10, batch_size=10)

# Convert the lists of training and testing accuracies to numpy arrays
```

```
train_acc_per_iteration_ql = np.array(train_acc_per_iteration_ql)
test_acc_per_iteration_ql = np.array(test_acc_per_iteration_ql)
```

```
For K = 5

For the EPOCH: 1  Training Accuracy = 0.5636  Testing Accuracy = 0.5822
For the EPOCH: 2  Training Accuracy = 0.7919666666666667  Testing Accuracy =
0.7997
For the EPOCH: 3  Training Accuracy = 0.7919666666666667  Testing Accuracy =
0.7997
For the EPOCH: 4  Training Accuracy = 0.8216333333333333  Testing Accuracy =
0.8207
For the EPOCH: 5  Training Accuracy = 0.8216333333333333  Testing Accuracy =
0.8207
For the EPOCH: 6  Training Accuracy = 0.8434333333333334  Testing Accuracy =
0.8442
For the EPOCH: 7  Training Accuracy = 0.8623166666666666  Testing Accuracy =
0.8621
For the EPOCH: 8  Training Accuracy = 0.8623166666666666  Testing Accuracy =
0.8621
For the EPOCH: 9  Training Accuracy = 0.875  Testing Accuracy = 0.8723
For the EPOCH: 10  Training Accuracy = 0.875  Testing Accuracy = 0.8723
```

[30]:
```
# Plot the training and testing accuracies as a function of training iterations
plt.figure(1)
plt.plot(train_acc_per_iteration_ql[:, 0], train_acc_per_iteration_ql[:, 1],␣
 ↪label='TRAIN ACCURACY')
plt.plot(test_acc_per_iteration_ql[:, 0], test_acc_per_iteration_ql[:,1],␣
 ↪label='TEST ACCURACY')
plt.title('TRAIN/TEST ACCURACIES FOR k = 5 (QUADRATIC LOSS)')
plt.xlabel('ITERATIONS')
plt.ylabel('ACCURACY')
plt.legend()
plt.grid()
plt.show()
```




[31]:
```
# Print the final testing accuracy achieved by the network
acc_5_ql = test_acc_per_iteration_ql[-1:,1]
print(f'Test Accuracy = {acc_5_ql*100} %')
```

```
Test Accuracy = [87.23] %
```

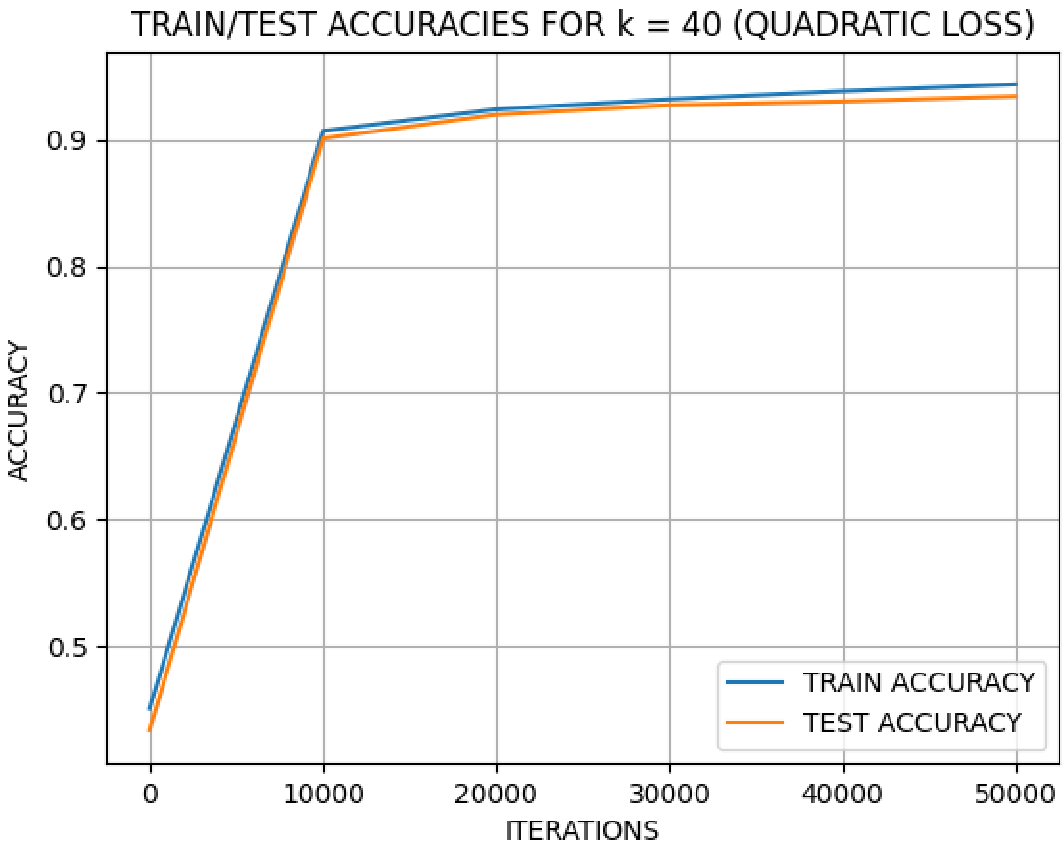
For k = 40

[32]:
```
print('For K = 40')

print()

# Call the function to train a shallow neural network with quadratic loss on␣
↪the input data
train_acc_per_iteration_ql, test_acc_per_iteration_ql =␣
↪shallow_neural_quadratic(x_train, x_test, y_train, y_test, lr=0.001, k=40,␣
↪epochs=10, batch_size=10)

# Convert the lists of training and testing accuracies to numpy arrays
train_acc_per_iteration_ql = np.array(train_acc_per_iteration_ql)
test_acc_per_iteration_ql = np.array(test_acc_per_iteration_ql)
```

```
For K = 40

For the EPOCH: 1  Training Accuracy = 0.45075  Testing Accuracy = 0.4336
For the EPOCH: 2  Training Accuracy = 0.9067666666666667  Testing Accuracy =
0.9009
For the EPOCH: 3  Training Accuracy = 0.9067666666666667  Testing Accuracy =
0.9009
For the EPOCH: 4  Training Accuracy = 0.92385  Testing Accuracy = 0.9197
For the EPOCH: 5  Training Accuracy = 0.92385  Testing Accuracy = 0.9197
For the EPOCH: 6  Training Accuracy = 0.9317  Testing Accuracy = 0.9271
For the EPOCH: 7  Training Accuracy = 0.9381  Testing Accuracy = 0.93
For the EPOCH: 8  Training Accuracy = 0.9381  Testing Accuracy = 0.93
For the EPOCH: 9  Training Accuracy = 0.9436333333333333  Testing Accuracy =
0.934
For the EPOCH: 10  Training Accuracy = 0.9436333333333333  Testing Accuracy =
0.934
```

[33]:

```
# Plot the training and testing accuracies as a function of training iterations
plt.figure(1)
plt.plot(train_acc_per_iteration_ql[:, 0], train_acc_per_iteration_ql[:, 1],␣
 ↪label='TRAIN ACCURACY')
plt.plot(test_acc_per_iteration_ql[:, 0], test_acc_per_iteration_ql[:,1],␣
 ↪label='TEST ACCURACY')
plt.title('TRAIN/TEST ACCURACIES FOR k = 40 (QUADRATIC LOSS)')
plt.xlabel('ITERATIONS')
plt.ylabel('ACCURACY')
plt.legend()
plt.grid()
plt.show()
```




[34]:
```
# Print the final testing accuracy achieved by the network
acc_40_ql = test_acc_per_iteration_ql[-1:,1]
print(f'Test Accuracy = {acc_40_ql*100}%')
```

```
Test Accuracy = [93.4]%
```

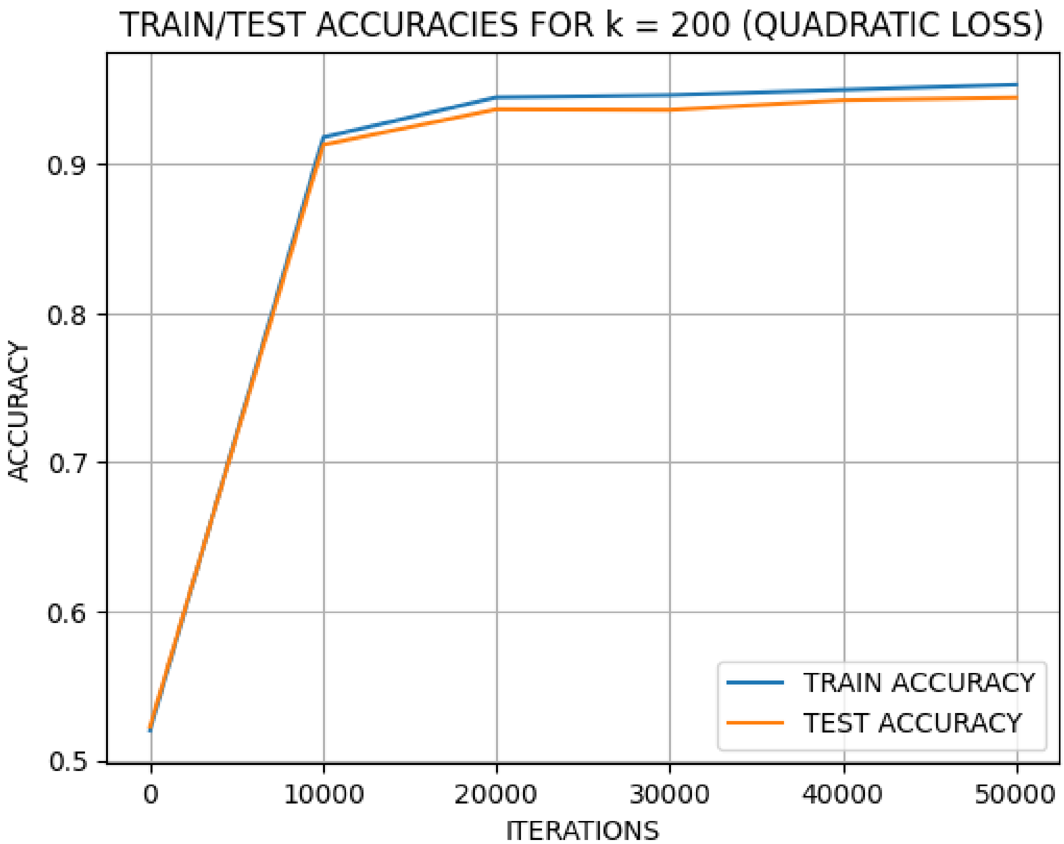
For k = 200

[35]:
```
print('For K = 200')

print()

# Call the function to train a shallow neural network with quadratic loss on␣
↪the input data
train_acc_per_iteration_ql, test_acc_per_iteration_ql =␣
↪shallow_neural_quadratic(x_train, x_test, y_train, y_test, lr=0.001, k=200,␣
↪epochs=10, batch_size=10)

# Convert the lists of training and testing accuracies to numpy arrays
train_acc_per_iteration_ql = np.array(train_acc_per_iteration_ql)
test_acc_per_iteration_ql = np.array(test_acc_per_iteration_ql)
```

```
For K = 200

For the EPOCH: 1  Training Accuracy = 0.5203666666666666  Testing Accuracy =
0.5228
For the EPOCH: 2  Training Accuracy = 0.9177666666666666  Testing Accuracy =
0.9128
For the EPOCH: 3  Training Accuracy = 0.9177666666666666  Testing Accuracy =
0.9128
For the EPOCH: 4  Training Accuracy = 0.94455  Testing Accuracy = 0.9365
For the EPOCH: 5  Training Accuracy = 0.94455  Testing Accuracy = 0.9365
For the EPOCH: 6  Training Accuracy = 0.9460833333333334  Testing Accuracy =
0.9361
For the EPOCH: 7  Training Accuracy = 0.94965  Testing Accuracy = 0.9426
For the EPOCH: 8  Training Accuracy = 0.94965  Testing Accuracy = 0.9426
For the EPOCH: 9  Training Accuracy = 0.9531833333333334  Testing Accuracy =
0.9444
For the EPOCH: 10  Training Accuracy = 0.9531833333333334  Testing Accuracy =
0.9444
```

[36]:

```python
# Plot the training and testing accuracies as a function of training iterations
plt.figure(1)
plt.plot(train_acc_per_iteration_ql[:, 0], train_acc_per_iteration_ql[:, 1],␣
 ↪label='TRAIN ACCURACY')
plt.plot(test_acc_per_iteration_ql[:, 0], test_acc_per_iteration_ql[:,1],␣
 ↪label='TEST ACCURACY')
plt.title('TRAIN/TEST ACCURACIES FOR k = 200 (QUADRATIC LOSS)')
plt.xlabel('ITERATIONS')
plt.ylabel('ACCURACY')
plt.legend()
plt.grid()
plt.show()
```




[37]:
```
# Print the final testing accuracy achieved by the network
acc_200_ql = test_acc_per_iteration_ql[-1:,1]
print(f'Test Accuracy = {acc_200_ql*100}%')
```

```
Test Accuracy = [94.44]%
```

**Comment on the role of hidden units k on the ease of optimization and accuracy.** *The number of Hidden Units allow the neural net to model the complex and non linear relationships between the inputs and outputs.*

*In this case as we observe, as k increases, the ease of optimization decreases due to the increase in number of hidden layer parameters. But the accuracy increases to some extent. It is to be noted that too much increase in k can also result in model overfitting*

#### 1.2.4 TASK 4: (7 pts)

Train a neural network classifier with logistic loss, namely $\ell(y, f(x)) = -ylog(\sigma(f(x))) - (1 - y)log(1 - \sigma(f(x)))$ where $\sigma(x) = 1/(1 + e - x)$ is the sigmoid function. Repeat step 3.

**Helper Functions**

[38]:
```
def sigmoid(z):
    """
    Apply the sigmoid function to the input.

    Input:
      z (numpy.ndarray): Input data.

    Returns:
      Output of applying the sigmoid function to the input.
    """
    # Apply the sigmoid function to the input and return the output
    return 1 / (1 + np.exp(-z))
```

[39]:
```
def logistic_loss(y_true, y_pred):
    """
    Compute the logistic loss between the true labels and predicted␣
↪probabilities.

    Input:
      y_true (numpy.ndarray): True labels.
      y_pred (numpy.ndarray): Predicted probabilities.

    Returns:
      Value of the logistic loss between the true labels and predicted␣
↪probabilities.
    """
    # Initialize a variable to accumulate the logistic loss
    cumulative_l = 0

    # Reshape the predicted probabilities array to be 1-dimensional
    y_pred = y_pred.reshape(-1,)

    # Compute the logistic loss for each pair of true label and predicted␣
↪probability
    for y, y_hat in zip(y_true, y_pred):
        # Compute the sigmoid function of the predicted probability
        sigmoid_y_hat = sigmoid(y_hat)

        # Compute the logistic loss for the current pair of true label and␣
↪predicted probability
        l = (y * np.log(sigmoid_y_hat)) + ((1 - y) * np.log(1 - sigmoid_y_hat))
        l = -l

        # Add the logistic loss to the cumulative loss
        cumulative_l += l
```

```
    # Compute the average logistic loss across all the pairs of true label and␣
↪predicted probability
    return cumulative_l / y_true.shape[0]
```

[40]:
```
def get_accuracy_ll(y_true, y_pred):
    """
    Compute the accuracy between the true labels and predicted probabilities.

    Input:
      y_true (numpy.ndarray): True labels.
      y_pred (numpy.ndarray): Predicted probabilities.

    Returns:
      float: Accuracy between the true labels and predicted probabilities.
    """
    # Reshape the predicted probabilities array to be 1-dimensional
    y_pred = y_pred.reshape(-1,)

    # Convert the predicted probabilities to binary values based on a threshold␣
↪of 0.5
    y_pred = np.where(y_pred > 0, 1, 0)

    # Compute the accuracy between the true labels and predicted binary values
    return np.sum(y_true == y_pred) / y_true.shape[0]
```

**Main Function**

[41]:
```
def shallow_neural_logistic(x_train, x_test, y_train, y_test, lr=0.01, k=5,␣
↪epochs=10, batch_size=10):
    """
    This function trains a shallow neural network with a logistic layer and␣
↪calculates training and testing accuracy.

    Input:
      x_train (numpy array): Training input data.
      x_test (numpy array): Testing input data.
      y_train (numpy array): Training output data.
      y_test (numpy array): Testing output data.
      lr (float, optional): Learning rate. Default is 0.01.
      k (int, optional): Number of hidden units. Default is 5.
      epochs (int, optional): Number of training epochs. Default is 10.
      batch_size (int, optional): Size of the mini-batch. Default is 10.

    Returns:
      A tuple containing two lists (train_acc_per_iteration_log,␣
↪test_acc_per_iteration_log) with the training and testing accuracy per␣
↪iteration.
```

```python
"""

# Set random seed for reproducibility
np.random.seed(112233)

# Initialize weights for the logistic layer
w_log = np.random.randn(x_train.shape[1], k) / np.sqrt(x_train.shape[1])
v_log = np.random.randn(k) / np.sqrt(k)

# Initialize lists to store training and testing accuracy per iteration
train_acc_per_iteration_log = []
test_acc_per_iteration_log = []

# Initialize iteration counter
iter_ctr_log = 0

# Start training loop
for epoch in range(epochs):

    # Shuffle the training data
    shuffled_indices = np.random.permutation(x_train.shape[0])
    x_shuffled = x_train[shuffled_indices]
    y_shuffled = y_train[shuffled_indices]

    i = 0

    # Process the data in mini-batches
    while i < x_train.shape[0]:

        # Get the current mini-batch
        x = x_shuffled[i:i+batch_size]
        y = y_shuffled[i:i+batch_size]

        # Perform forward pass
        z1 = np.matmul(x, w_log)
        y1 = relu(z1)
        z2 = np.matmul(y1, v_log)
        y2 = np.round(z2)

        # Calculate error
        delta_2 = (y2 - y)

        # Calculate weight update for output layer
        dv = np.matmul(y1.T, delta_2) / x.shape[0]

        # Calculate weight update for hidden layer
        relu_derivative = relu_deriv(z1)
```

```
            delta_1 = np.matmul(delta_2.reshape(-1,1), v_log.reshape(-1,1).T) *␣
↪relu_derivative
            dw = np.matmul(x.T, delta_1) / x.shape[0]

            # Update weights
            w_log -= lr*dw
            v_log -= lr*dv

            # Increment mini-batch counter
            i += batch_size

            # Calculate training and testing accuracy every 10000 iterations or␣
↪at the start
            if iter_ctr_log == 0 or iter_ctr_log % 10000 == 0:
                z1 = np.matmul(x_train, w_log)
                y1 = relu(z1)
                z2 = np.matmul(y1, v_log)
                y2 = np.round(z2)
                train_accuracy_log = get_accuracy_ql(y_train, y2)
                z1 = np.matmul(x_test, w_log)
                y1 = relu(z1)
                z2 = np.matmul(y1, v_log)
                y2 = np.round(z2)
                test_accuracy_log = get_accuracy_ql(y_test, y2)

                # Append training and testing accuracy to their respective lists
                train_acc_per_iteration_log.append((iter_ctr_log,␣
↪train_accuracy_log))
                test_acc_per_iteration_log.append((iter_ctr_log,␣
↪test_accuracy_log))

            # Increment iteration counter
            iter_ctr_log += 1

        # Print training and testing accuracy for the current epoch
        print('For the EPOCH:', epoch + 1, ' Training Accuracy =',␣
↪train_accuracy_log, ' Testing Accuracy =', test_accuracy_log)

    # Return training and testing accuracy per iteration
    return train_acc_per_iteration_log, test_acc_per_iteration_log
```

### Testing the Shallow Net with Logistic Loss

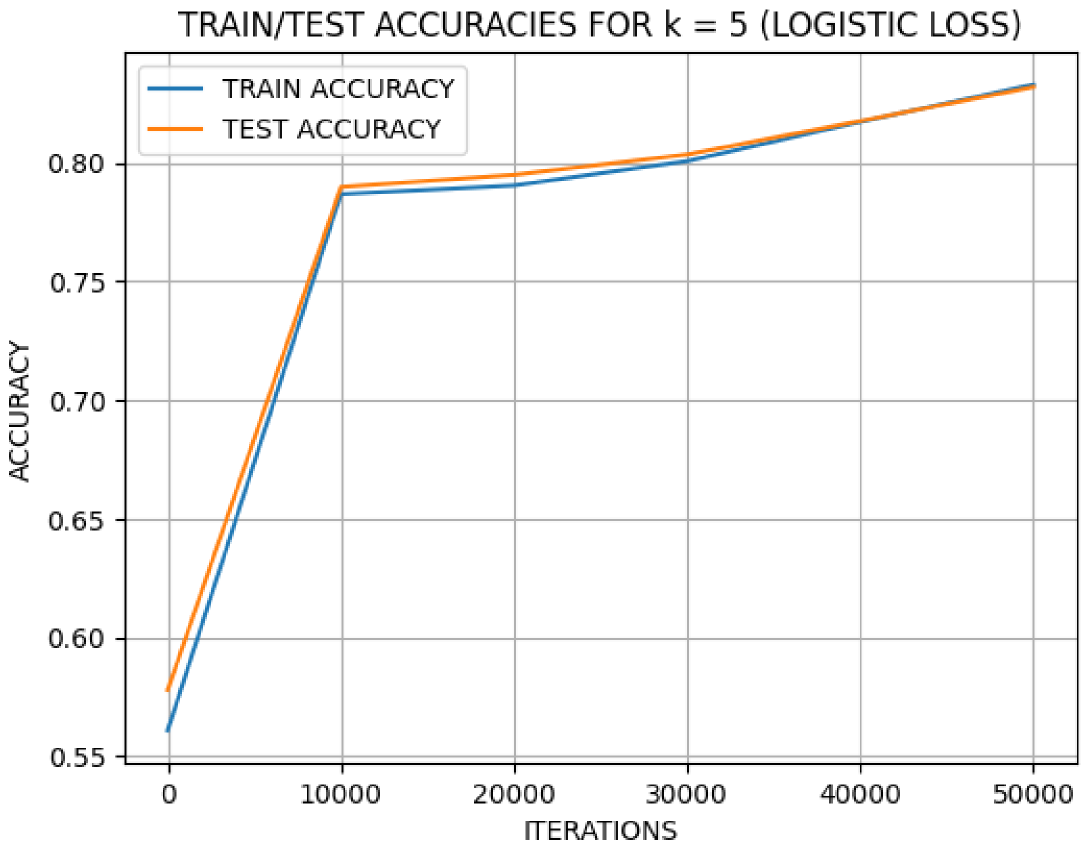
For k = 5

[42]:
```
print ('For K = 5')
```

```
print()

# Call the function to train a shallow neural network with Logistic loss on the␣
↪input data
train_acc_per_iteration_ll, test_acc_per_iteration_ll =␣
↪shallow_neural_logistic(x_train, x_test, y_train, y_test, lr=0.001, k=5,␣
↪epochs=10, batch_size=10)

# Convert the lists of training and testing accuracies to numpy arrays
train_acc_per_iteration_ll = np.array(train_acc_per_iteration_ll)
test_acc_per_iteration_ll = np.array(test_acc_per_iteration_ll)
```

```
For K = 5

For the EPOCH: 1  Training Accuracy = 0.5609  Testing Accuracy = 0.5779
For the EPOCH: 2  Training Accuracy = 0.7867666666666666  Testing Accuracy =
0.7898
For the EPOCH: 3  Training Accuracy = 0.7867666666666666  Testing Accuracy =
0.7898
For the EPOCH: 4  Training Accuracy = 0.7904333333333333  Testing Accuracy =
0.7949
For the EPOCH: 5  Training Accuracy = 0.7904333333333333  Testing Accuracy =
0.7949
For the EPOCH: 6  Training Accuracy = 0.8007  Testing Accuracy = 0.8035
For the EPOCH: 7  Training Accuracy = 0.8172333333333334  Testing Accuracy =
0.8176
For the EPOCH: 8  Training Accuracy = 0.8172333333333334  Testing Accuracy =
0.8176
For the EPOCH: 9  Training Accuracy = 0.8329666666666666  Testing Accuracy =
0.8319
For the EPOCH: 10  Training Accuracy = 0.8329666666666666  Testing Accuracy =
0.8319
```

[43]:
```
# Plot the training and testing accuracies as a function of training iterations
plt.figure(1)
plt.plot(train_acc_per_iteration_ll[:, 0], train_acc_per_iteration_ll[:, 1],␣
↪label='TRAIN ACCURACY')
plt.plot(test_acc_per_iteration_ll[:, 0], test_acc_per_iteration_ll[:,1],␣
↪label='TEST ACCURACY')
plt.title('TRAIN/TEST ACCURACIES FOR k = 5 (LOGISTIC LOSS)')
plt.xlabel('ITERATIONS')
plt.ylabel('ACCURACY')
plt.legend()
plt.grid()
plt.show()
```




[44]:
```
# Print the final testing accuracy achieved by the network
acc_5_ql = test_acc_per_iteration_ll[-1:,1]
print(f'Test Accuracy = {acc_5_ql*100}%')
```

```
Test Accuracy = [83.19]%
```

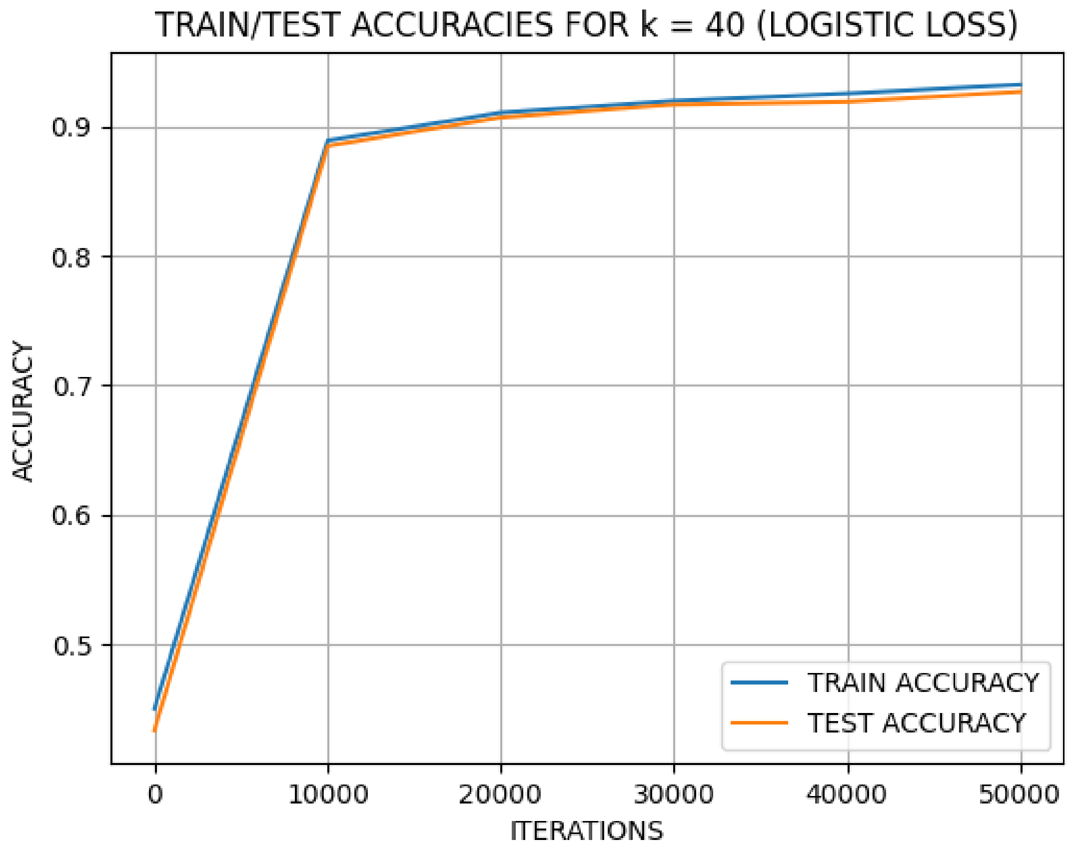
For k = 40

[45]:
```
print ('For K = 40')

print()
# Call the function to train a shallow neural network with Logistic loss on the␣
↪input data
train_acc_per_iteration_ll, test_acc_per_iteration_ll =␣
↪shallow_neural_logistic(x_train, x_test, y_train, y_test, lr=0.001, k=40,␣
↪epochs=10, batch_size=10)

# Convert the lists of training and testing accuracies to numpy arrays
train_acc_per_iteration_ll = np.array(train_acc_per_iteration_ll)
test_acc_per_iteration_ll = np.array(test_acc_per_iteration_ll)
```

```
For K = 40

For the EPOCH: 1  Training Accuracy = 0.44966666666666666  Testing Accuracy =
0.433
For the EPOCH: 2  Training Accuracy = 0.8891833333333333  Testing Accuracy =
0.885
For the EPOCH: 3  Training Accuracy = 0.8891833333333333  Testing Accuracy =
0.885
For the EPOCH: 4  Training Accuracy = 0.9107  Testing Accuracy = 0.907
For the EPOCH: 5  Training Accuracy = 0.9107  Testing Accuracy = 0.907
For the EPOCH: 6  Training Accuracy = 0.9198166666666666  Testing Accuracy =
0.917
For the EPOCH: 7  Training Accuracy = 0.9255166666666667  Testing Accuracy =
0.9192
For the EPOCH: 8  Training Accuracy = 0.9255166666666667  Testing Accuracy =
0.9192
For the EPOCH: 9  Training Accuracy = 0.9325  Testing Accuracy = 0.9269
For the EPOCH: 10  Training Accuracy = 0.9325  Testing Accuracy = 0.9269
```

[46]:

```python
# Plot the training and testing accuracies as a function of training iterations
plt.figure(1)
plt.plot(train_acc_per_iteration_ll[:, 0], train_acc_per_iteration_ll[:, 1],␣
 ↪label='TRAIN ACCURACY')
plt.plot(test_acc_per_iteration_ll[:, 0], test_acc_per_iteration_ll[:,1],␣
 ↪label='TEST ACCURACY')
plt.title('TRAIN/TEST ACCURACIES FOR k = 40 (LOGISTIC LOSS)')
plt.xlabel('ITERATIONS')
plt.ylabel('ACCURACY')
plt.legend()
plt.grid()
plt.show()
```




[47]:
```
# Print the final testing accuracy achieved by the network
acc_40_ql = test_acc_per_iteration_ll[-1:,1]
print(f'Test Accuracy = {acc_40_ql*100}%')
```

```
Test Accuracy = [92.69]%
```

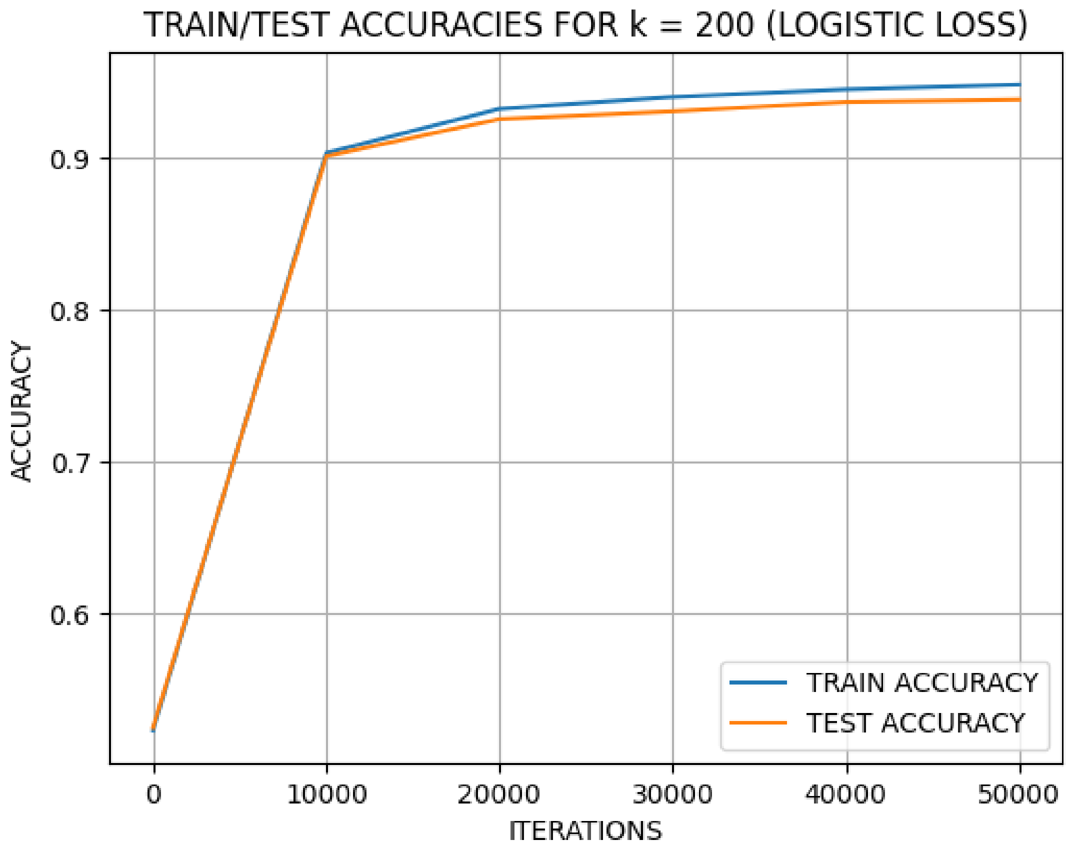
For k = 200

[48]:
```
print ('For K = 200')

print()

# Call the function to train a shallow neural network with Logistic loss on the␣
↪input data
train_acc_per_iteration_ll, test_acc_per_iteration_ll =␣
↪shallow_neural_logistic(x_train, x_test, y_train, y_test, lr=0.001, k=200,␣
↪epochs=10, batch_size=10)

# Convert the lists of training and testing accuracies to numpy arrays
train_acc_per_iteration_ll = np.array(train_acc_per_iteration_ll)
test_acc_per_iteration_ll = np.array(test_acc_per_iteration_ll)
```

```
For K = 200

For the EPOCH: 1  Training Accuracy = 0.5234333333333333  Testing Accuracy =
0.5252
For the EPOCH: 2  Training Accuracy = 0.9028666666666667  Testing Accuracy =
0.9008
For the EPOCH: 3  Training Accuracy = 0.9028666666666667  Testing Accuracy =
0.9008
For the EPOCH: 4  Training Accuracy = 0.9318333333333333  Testing Accuracy =
0.9251
For the EPOCH: 5  Training Accuracy = 0.9318333333333333  Testing Accuracy =
0.9251
For the EPOCH: 6  Training Accuracy = 0.9396333333333333  Testing Accuracy =
0.9302
For the EPOCH: 7  Training Accuracy = 0.9446333333333333  Testing Accuracy =
0.9363
For the EPOCH: 8  Training Accuracy = 0.9446333333333333  Testing Accuracy =
0.9363
For the EPOCH: 9  Training Accuracy = 0.9478333333333333  Testing Accuracy =
0.9379
For the EPOCH: 10  Training Accuracy = 0.9478333333333333  Testing Accuracy =
0.9379
```

[49]:

```
# Plot the training and testing accuracies as a function of training iterations
plt.figure(1)
plt.plot(train_acc_per_iteration_ll[:, 0], train_acc_per_iteration_ll[:, 1],␣
 ↪label='TRAIN ACCURACY')
plt.plot(test_acc_per_iteration_ll[:, 0], test_acc_per_iteration_ll[:,1],␣
 ↪label='TEST ACCURACY')
plt.title('TRAIN/TEST ACCURACIES FOR k = 200 (LOGISTIC LOSS)')
plt.xlabel('ITERATIONS')
plt.ylabel('ACCURACY')
plt.legend()
plt.grid()
plt.show()
```




[50]:
```
# Print the final testing accuracy achieved by the network
acc_200_ql = test_acc_per_iteration_ll[-1:,1]
print(f'Test Accuracy = {acc_200_ql*100}%')
```

```
Test Accuracy = [93.79]%
```

**Comment on the role of hidden units k on the ease of optimization and accuracy.** *Similar to the previous case, here also it is observed that, as k increases, the ease of optimization decreases due to the increase in number of hidden layer parameters. But the accuracy increases to some extent.*

#### 1.2.5 TASK 5: (2 pts)

Comment on the difference between linear model and neural net. Comment on the differences between logistic and quadratic loss in terms of optimization and test/train accuracy.

*Linear models are simpler as they have a linear relationship between the inputs and the output making them suitable and a well-performing option for simpler problems. Neural Net on the other hand, is capable enough to handle complex regression/classification problems while giving a better performance than the Linear models even on simpler tasks.*

*Though my implementation somehow did not perform as expected. The neural net with logistic loss should have performed better than the one with quadratic loss in terms if train/test accuracy, as logistic losses are more resistant to outliers and have a smooth surface thereby eliminating the scenarios of multiple local minimas.*

### 1.3 Submission

[51]:
```
!sudo apt-get update
!sudo apt-get install texlive-xetex texlive-fonts-recommended
```

```
Hit:1 http://archive.ubuntu.com/ubuntu focal InRelease
Get:2 http://archive.ubuntu.com/ubuntu focal-updates InRelease [114 kB]
Get:3 https://cloud.r-project.org/bin/linux/ubuntu focal-cran40/ InRelease
[3,622 B]
Get:4 https://developer.download.nvidia.com/compute/cuda/repos/ubuntu2004/x86_64
InRelease [1,581 B]
Hit:5 http://ppa.launchpad.net/c2d4u.team/c2d4u4.0+/ubuntu focal InRelease
Get:6 http://security.ubuntu.com/ubuntu focal-security InRelease [114 kB]
Get:7 http://archive.ubuntu.com/ubuntu focal-backports InRelease [108 kB]
Hit:8 http://ppa.launchpad.net/cran/libgit2/ubuntu focal InRelease
Get:9 https://developer.download.nvidia.com/compute/cuda/repos/ubuntu2004/x86_64
Packages [1,009 kB]
Get:10 http://archive.ubuntu.com/ubuntu focal-updates/main amd64 Packages [3,150
kB]
Hit:11 http://ppa.launchpad.net/deadsnakes/ppa/ubuntu focal InRelease
Hit:12 http://ppa.launchpad.net/graphics-drivers/ppa/ubuntu focal InRelease
Hit:13 http://ppa.launchpad.net/ubuntugis/ppa/ubuntu focal InRelease
Get:14 http://security.ubuntu.com/ubuntu focal-security/main amd64 Packages
[2,669 kB]
Fetched 7,169 kB in 2s (3,381 kB/s)
Reading package lists… Done
Reading package lists… Done
Building dependency tree
Reading state information… Done
The following additional packages will be installed:
  dvisvgm fonts-droid-fallback fonts-lato fonts-lmodern fonts-noto-mono
  fonts-texgyre fonts-urw-base35 javascript-common libapache-pom-java
  libcommons-logging-java libcommons-parent-java libfontbox-java libfontenc1
  libgs9 libgs9-common libharfbuzz-icu0 libidn11 libijs-0.35 libjbig2dec0
  libjs-jquery libkpathsea6 libpdfbox-java libptexenc1 libruby2.7 libsynctex2
  libteckit0 libtexlua53 libtexluajit2 libwoff1 libzzip-0-13 lmodern
  poppler-data preview-latex-style rake ruby ruby-minitest ruby-net-telnet
  ruby-power-assert ruby-test-unit ruby-xmlrpc ruby2.7 rubygems-integration
  t1utils teckit tex-common tex-gyre texlive-base texlive-binaries
  texlive-latex-base texlive-latex-extra texlive-latex-recommended
  texlive-pictures texlive-plain-generic tipa xfonts-encodings xfonts-utils
Suggested packages:
  fonts-noto fonts-freefont-otf | fonts-freefont-ttf apache2 | lighttpd
```

```
  | httpd libavalon-framework-java libcommons-logging-java-doc
  libexcalibur-logkit-java liblog4j1.2-java poppler-utils ghostscript
  fonts-japanese-mincho | fonts-ipafont-mincho fonts-japanese-gothic
  | fonts-ipafont-gothic fonts-arphic-ukai fonts-arphic-uming fonts-nanum ri
  ruby-dev bundler debhelper gv | postscript-viewer perl-tk xpdf | pdf-viewer
  xzdec texlive-fonts-recommended-doc texlive-latex-base-doc python3-pygments
  icc-profiles libfile-which-perl libspreadsheet-parseexcel-perl
  texlive-latex-extra-doc texlive-latex-recommended-doc texlive-luatex
  texlive-pstricks dot2tex prerex ruby-tcltk | libtcltk-ruby
  texlive-pictures-doc vprerex default-jre-headless
The following NEW packages will be installed:
  dvisvgm fonts-droid-fallback fonts-lato fonts-lmodern fonts-noto-mono
  fonts-texgyre fonts-urw-base35 javascript-common libapache-pom-java
  libcommons-logging-java libcommons-parent-java libfontbox-java libfontenc1
  libgs9 libgs9-common libharfbuzz-icu0 libidn11 libijs-0.35 libjbig2dec0
  libjs-jquery libkpathsea6 libpdfbox-java libptexenc1 libruby2.7 libsynctex2
  libteckit0 libtexlua53 libtexluajit2 libwoff1 libzzip-0-13 lmodern
  poppler-data preview-latex-style rake ruby ruby-minitest ruby-net-telnet
  ruby-power-assert ruby-test-unit ruby-xmlrpc ruby2.7 rubygems-integration
  t1utils teckit tex-common tex-gyre texlive-base texlive-binaries
  texlive-fonts-recommended texlive-latex-base texlive-latex-extra
  texlive-latex-recommended texlive-pictures texlive-plain-generic
  texlive-xetex tipa xfonts-encodings xfonts-utils
0 upgraded, 58 newly installed, 0 to remove and 25 not upgraded.
Need to get 169 MB of archives.
After this operation, 537 MB of additional disk space will be used.
Get:1 http://archive.ubuntu.com/ubuntu focal/main amd64 fonts-droid-fallback all
1:6.0.1r16-1.1 [1,805 kB]
Get:2 http://archive.ubuntu.com/ubuntu focal/main amd64 fonts-lato all 2.0-2
[2,698 kB]
Get:3 http://archive.ubuntu.com/ubuntu focal/main amd64 poppler-data all 0.4.9-2
[1,475 kB]
Get:4 http://archive.ubuntu.com/ubuntu focal/universe amd64 tex-common all 6.13
[32.7 kB]
Get:5 http://archive.ubuntu.com/ubuntu focal/main amd64 fonts-urw-base35 all
20170801.1-3 [6,333 kB]
Get:6 http://archive.ubuntu.com/ubuntu focal-updates/main amd64 libgs9-common
all 9.50~dfsg-5ubuntu4.7 [681 kB]
Get:7 http://archive.ubuntu.com/ubuntu focal/main amd64 libidn11 amd64
1.33-2.2ubuntu2 [46.2 kB]
Get:8 http://archive.ubuntu.com/ubuntu focal/main amd64 libijs-0.35 amd64
0.35-15 [15.7 kB]
Get:9 http://archive.ubuntu.com/ubuntu focal/main amd64 libjbig2dec0 amd64
0.18-1ubuntu1 [60.0 kB]
Get:10 http://archive.ubuntu.com/ubuntu focal-updates/main amd64 libgs9 amd64
9.50~dfsg-5ubuntu4.7 [2,173 kB]
Get:11 http://archive.ubuntu.com/ubuntu focal/main amd64 libkpathsea6 amd64
2019.20190605.51237-3build2 [57.0 kB]
```

```
Get:12 http://archive.ubuntu.com/ubuntu focal/main amd64 libwoff1 amd64
1.0.2-1build2 [42.0 kB]
Get:13 http://archive.ubuntu.com/ubuntu focal/universe amd64 dvisvgm amd64
2.8.1-1build1 [1,048 kB]
Get:14 http://archive.ubuntu.com/ubuntu focal/universe amd64 fonts-lmodern all
2.004.5-6 [4,532 kB]
Get:15 http://archive.ubuntu.com/ubuntu focal-updates/main amd64 fonts-noto-mono
all 20200323-1build1~ubuntu20.04.1 [80.6 kB]
Get:16 http://archive.ubuntu.com/ubuntu focal/universe amd64 fonts-texgyre all
20180621-3 [10.2 MB]
Get:17 http://archive.ubuntu.com/ubuntu focal/main amd64 javascript-common all
11 [6,066 B]
Get:18 http://archive.ubuntu.com/ubuntu focal/universe amd64 libapache-pom-java
all 18-1 [4,720 B]
Get:19 http://archive.ubuntu.com/ubuntu focal/universe amd64 libcommons-parent-
java all 43-1 [10.8 kB]
Get:20 http://archive.ubuntu.com/ubuntu focal/universe amd64 libcommons-logging-
java all 1.2-2 [60.3 kB]
Get:21 http://archive.ubuntu.com/ubuntu focal/main amd64 libfontenc1 amd64
1:1.1.4-0ubuntu1 [14.0 kB]
Get:22 http://archive.ubuntu.com/ubuntu focal-updates/main amd64 libharfbuzz-
icu0 amd64 2.6.4-1ubuntu4.2 [5,580 B]
Get:23 http://archive.ubuntu.com/ubuntu focal/main amd64 libjs-jquery all
3.3.1~dfsg-3 [329 kB]
Get:24 http://archive.ubuntu.com/ubuntu focal/main amd64 libptexenc1 amd64
2019.20190605.51237-3build2 [35.5 kB]
Get:25 http://archive.ubuntu.com/ubuntu focal/main amd64 rubygems-integration
all 1.16 [5,092 B]
Get:26 http://archive.ubuntu.com/ubuntu focal-updates/main amd64 ruby2.7 amd64
2.7.0-5ubuntu1.10 [95.6 kB]
Get:27 http://archive.ubuntu.com/ubuntu focal/main amd64 ruby amd64 1:2.7+1
[5,412 B]
Get:28 http://archive.ubuntu.com/ubuntu focal/main amd64 rake all 13.0.1-4 [61.6
kB]
Get:29 http://archive.ubuntu.com/ubuntu focal/main amd64 ruby-minitest all
5.13.0-1 [40.9 kB]
Get:30 http://archive.ubuntu.com/ubuntu focal/main amd64 ruby-net-telnet all
0.1.1-2 [12.6 kB]
Get:31 http://archive.ubuntu.com/ubuntu focal/main amd64 ruby-power-assert all
1.1.7-1 [11.4 kB]
Get:32 http://archive.ubuntu.com/ubuntu focal/main amd64 ruby-test-unit all
3.3.5-1 [73.2 kB]
Get:33 http://archive.ubuntu.com/ubuntu focal/main amd64 ruby-xmlrpc all 0.3.0-2
[23.8 kB]
Get:34 http://archive.ubuntu.com/ubuntu focal-updates/main amd64 libruby2.7
amd64 2.7.0-5ubuntu1.10 [3,532 kB]
Get:35 http://archive.ubuntu.com/ubuntu focal/main amd64 libsynctex2 amd64
2019.20190605.51237-3build2 [55.0 kB]
```

```
Get:36 http://archive.ubuntu.com/ubuntu focal/universe amd64 libteckit0 amd64
2.5.8+ds2-5ubuntu2 [320 kB]
Get:37 http://archive.ubuntu.com/ubuntu focal/main amd64 libtexlua53 amd64
2019.20190605.51237-3build2 [105 kB]
Get:38 http://archive.ubuntu.com/ubuntu focal/main amd64 libtexluajit2 amd64
2019.20190605.51237-3build2 [235 kB]
Get:39 http://archive.ubuntu.com/ubuntu focal/universe amd64 libzzip-0-13 amd64
0.13.62-3.2ubuntu1 [26.2 kB]
Get:40 http://archive.ubuntu.com/ubuntu focal/main amd64 xfonts-encodings all
1:1.0.5-0ubuntu1 [573 kB]
Get:41 http://archive.ubuntu.com/ubuntu focal/main amd64 xfonts-utils amd64
1:7.7+6 [91.5 kB]
Get:42 http://archive.ubuntu.com/ubuntu focal/universe amd64 lmodern all
2.004.5-6 [9,474 kB]
Get:43 http://archive.ubuntu.com/ubuntu focal/universe amd64 preview-latex-style
all 11.91-2ubuntu2 [184 kB]
Get:44 http://archive.ubuntu.com/ubuntu focal/main amd64 t1utils amd64 1.41-3
[56.1 kB]
Get:45 http://archive.ubuntu.com/ubuntu focal/universe amd64 teckit amd64
2.5.8+ds2-5ubuntu2 [687 kB]
Get:46 http://archive.ubuntu.com/ubuntu focal/universe amd64 tex-gyre all
20180621-3 [6,209 kB]
Get:47 http://archive.ubuntu.com/ubuntu focal/universe amd64 texlive-binaries
amd64 2019.20190605.51237-3build2 [8,041 kB]
Get:48 http://archive.ubuntu.com/ubuntu focal/universe amd64 texlive-base all
2019.20200218-1 [20.8 MB]
Get:49 http://archive.ubuntu.com/ubuntu focal/universe amd64 texlive-fonts-
recommended all 2019.20200218-1 [4,972 kB]
Get:50 http://archive.ubuntu.com/ubuntu focal/universe amd64 texlive-latex-base
all 2019.20200218-1 [990 kB]
Get:51 http://archive.ubuntu.com/ubuntu focal/universe amd64 libfontbox-java all
1:1.8.16-2 [207 kB]
Get:52 http://archive.ubuntu.com/ubuntu focal/universe amd64 libpdfbox-java all
1:1.8.16-2 [5,199 kB]
Get:53 http://archive.ubuntu.com/ubuntu focal/universe amd64 texlive-latex-
recommended all 2019.20200218-1 [15.7 MB]
Get:54 http://archive.ubuntu.com/ubuntu focal/universe amd64 texlive-pictures
all 2019.20200218-1 [4,492 kB]
Get:55 http://archive.ubuntu.com/ubuntu focal/universe amd64 texlive-latex-extra
all 2019.202000218-1 [12.5 MB]
Get:56 http://archive.ubuntu.com/ubuntu focal/universe amd64 texlive-plain-
generic all 2019.202000218-1 [24.6 MB]
79% [56 texlive-plain-generic 13.8 kB/24.6 MB 0%]                 7,608 kB/s
5s^C
```

[52]:
```
!jupyter nbconvert --log-level CRITICAL --to pdf Nityash_Gautam_Assignment_2.
↪ipynb # make sure the ipynb name is correct
```

```
Traceback (most recent call last):
  File "/usr/local/bin/jupyter-nbconvert", line 8, in <module>
    sys.exit(main())
  File "/usr/local/lib/python3.10/dist-packages/jupyter_core/application.py",
line 277, in launch_instance
    return super().launch_instance(argv=argv, **kwargs)
  File "/usr/local/lib/python3.10/dist-
packages/traitlets/config/application.py", line 992, in launch_instance
    app.start()
  File "/usr/local/lib/python3.10/dist-packages/nbconvert/nbconvertapp.py", line
423, in start
    self.convert_notebooks()
  File "/usr/local/lib/python3.10/dist-packages/nbconvert/nbconvertapp.py", line
597, in convert_notebooks
    self.convert_single_notebook(notebook_filename)
  File "/usr/local/lib/python3.10/dist-packages/nbconvert/nbconvertapp.py", line
560, in convert_single_notebook
    output, resources = self.export_single_notebook(
  File "/usr/local/lib/python3.10/dist-packages/nbconvert/nbconvertapp.py", line
488, in export_single_notebook
    output, resources = self.exporter.from_filename(
  File "/usr/local/lib/python3.10/dist-
packages/nbconvert/exporters/exporter.py", line 189, in from_filename
    return self.from_file(f, resources=resources, **kw)
  File "/usr/local/lib/python3.10/dist-
packages/nbconvert/exporters/exporter.py", line 206, in from_file
    return self.from_notebook_node(
  File "/usr/local/lib/python3.10/dist-packages/nbconvert/exporters/pdf.py",
line 194, in from_notebook_node
    self.run_latex(tex_file)
  File "/usr/local/lib/python3.10/dist-packages/nbconvert/exporters/pdf.py",
line 164, in run_latex
    return self.run_command(
  File "/usr/local/lib/python3.10/dist-packages/nbconvert/exporters/pdf.py",
line 111, in run_command
    raise OSError(
OSError: xelatex not found on PATH, if you have not installed xelatex you may
need to do so. Find further instructions at
https://nbconvert.readthedocs.io/en/latest/install.html#installing-tex.
```