CS178 Homework 3

Due: Monday October 6 2023 (11:59pm)

Instructions

This homework (and subsequent ones) will involve data analysis and reporting on methods and results using Python code. You will submit a **single PDF file** that contains everything to Gradescope. This includes any text you wish to include to describe your results, the complete code snippets of how you attempted each problem, any figures that were generated, and scans of any work on paper that you wish to include. It is important that you include enough detail that we know how you solved the problem, since otherwise we will be unable to grade it.

Your homeworks will be given to you as Jupyter notebooks containing the problem descriptions and some template code that will help you get started. You are encouraged to use these starter Jupyter notebooks to complete your assignment and to write your report. This will help you not only ensure that all of the code for the solutions is included, but also will provide an easy way to export your results to a PDF file (for example, doing *print preview* and *printing to pdf*). I recommend liberal use of Markdown cells to create headers for each problem and sub-problem, explaining your implementation/answers, and including any mathematical equations. For parts of the homework you do on paper, scan it in such that it is legible (there are a number of free Android/iOS scanning apps, if you do not have access to a scanner), and include it as an image in the Jupyter notebook.

Double check that all of your answers are legible on Gradescope, e.g. make sure any text you have written does not get cut off.

If you have any questions/concerns about using Jupyter notebooks, ask us on EdD. If you decide not to use Jupyter notebooks, but go with Microsoft Word or LaTeX to create your PDF file, make sure that all of the answers can be generated from the code snippets included in the document.

Summary of Assignment: 100 total points

- Problem 1: A Small Neural Network (25 points)
  - Problem 1.1: Forward Pass (10 points)
  - Problem 1.2: Evaluate Loss (10 points)
  - Problem 1.3: Network Size (5 points)
- Problem 2: Logistic Regression (35 points)
  - Problem 2.1: Initial Training (10 points)
  - Problem 2.2: Regularization (10 points)
  - Problem 2.3: Interpreting the Weights (5 points)
  - Problem 2.4: Learning Curves (10 points)
- Problem 3: Neural Networks in Code (35 points)
  - Problem 2.1: Varying the Amount of Training Data (15 points)
  - Problem 2.3: Optimization Curves (10 points)
  - Problem 2.3: Tuning your Neural Network (10 points)
- Statement of Collaboration (5 points)

Before we get started, let's import some libraries that you will make use of in this assignment. Make sure that you run the code cell below in order to import these libraries.

Important: In the code block below, we set `seed=1234`. This is to ensure your code has reproducible results and is important for grading. Do not change this. If you are not using the provided Jupyter notebook, make sure to also set the random seed as below.

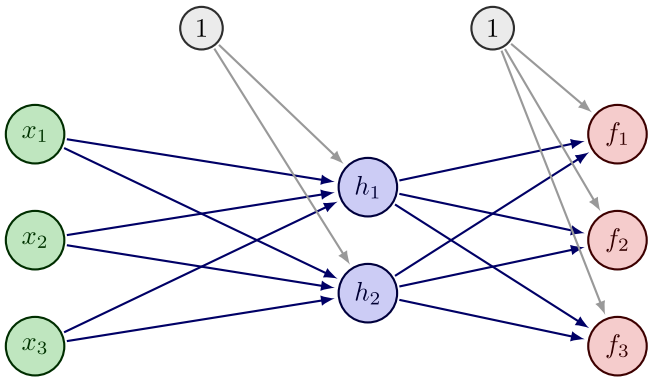
Important: Do not change any codes we give you below, except for those waiting for you to complete. This is to ensure your code has reproducible results and is important for grading.

In [18]:
```
import numpy as np
import matplotlib.pyplot as plt

from sklearn.datasets import fetch_openml           # common data set access
from sklearn.preprocessing import StandardScaler    # scaling transform
from sklearn.model_selection import train_test_split # validation tools
from sklearn.metrics import accuracy_score, zero_one_loss

from sklearn.linear_model import LogisticRegression
from sklearn.neural_network import MLPClassifier

import warnings
warnings.filterwarnings('ignore')

# Fix the random seed for reproducibility
# !! Important !! : do not change this
```

```
seed = 1234
np.random.seed(seed)
```

Problem 1: A Small Neural Network

Consider the small neural network given in the image below, which will classify a 3-dimensional feature vector $\mathbf{x}$ into one of three classes ($y = 0, 1, 2$):



You are given an input to this network $\mathbf{x}$, $$ \mathbf{x} = \begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix} = \begin{bmatrix} 1 & 3 & -2 \end{bmatrix} $$ as well as weights W for the hidden layer and weights B for the output layer.

$$ W = \begin{bmatrix} w_{01} & w_{11} & w_{21} & w_{31} \\ w_{02} & w_{12} & w_{22} & w_{32} \\ \end{bmatrix} = \begin{bmatrix} 1 & -1 & 0 & 5 \\ 2 & 1 & 1 & 2 \\ \end{bmatrix} $$

$$ B = \begin{bmatrix} \beta_{01} & \beta_{11} & \beta_{21} \\ \beta_{02} & \beta_{12} & \beta_{22} \\ \beta_{03} & \beta_{13} & \beta_{23} \\ \end{bmatrix} = \begin{bmatrix} 4 & -1 & 0 \\ 3 & 0 & 2 \\ 2 & 1 & 1 \end{bmatrix} $$

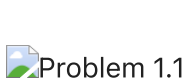
For example, w_{12} is the weight connecting input x_1 to hidden node h_2; w_{01} is the constant (bias) term for h_1, etc.

This network uses the ReLU activation function for the hidden layer, and uses the softmax activation function for the output layer.

Answer the following questions about this network.

Problem 1.1 (10 points): Forward Pass

- Given the inputs and weights above, compute the values of the hidden units h_1, h_2 and the outputs f_0, f_1, f_2. You should do this by hand, i.e. you should not write any code to do the calculation, but feel free to use a calculator to help you do the computations.
- You can optionally use $\LaTeX$ in your answer on the Jupyter notebook. Otherwise, write your answer on paper and include a picture of your answer in this notebook. In order to include an image in Jupyter notebook, save the image in the same directory as the .ipynb file and then write `![caption](image.png)`. Alternatively, you may go to Edit --> Insert Image at the top menu to insert an image into a Markdown cell. **Double check that your image is visible in your PDF submission.**
- What class would the network predict for the input $\mathbf{x}$?



Given that python is 0-indexed, the network would predict **class 1** for input x as the value of f1 is the highest (0.909). Therefore, Class 1 is the best match for input x.

Problem 1.2 (10 points): Evaluate Loss

Typically when we train neural networks for classification, we seek to minimize the log-loss function. Note that the output of the log-loss function is always nonnegative (≥ 0), but can be arbitrarily large (you should pause for a second and make sure you understand why this is true).

- Suppose the true label for the input $\mathbf{x}$ is $y = 1$. What would be the value of our loss function based on the network's prediction for $\mathbf{x}$?
- Suppose instead that the true label for the input $\mathbf{x}$ is $y = 2$. What would be the value of our loss function based on the network's prediction for $\mathbf{x}$?

You are free to use numpy / Python to help you calculate this, but don't use any neural network libraries that will automatically calculate the loss for you.

In [19]:
```
import numpy as np

# Loss for true label y = 1
loss_y1 = -np.sum(np.array([0, 1, 0]) * np.log([0.045, 0.909, 0.045]))
```

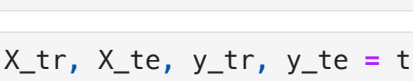
```
# Loss for true label y = 2
loss_y2 = -np.sum(np.array([0, 0, 1]) * np.log([0.045, 0.909, 0.045]))

print(f"Loss when true label y=1: {loss_y1}")
print(f"Loss when true label y=2: {loss_y2}")
```

```
Loss when true label y=1: 0.09541018480465818
Loss when true label y=2: 3.101092789211817
```

Problem 1.3 (5 points): Network Size

- Suppose we change our network so that there are 12 hidden nodes instead of 2. How many total parameters (weights and biases) are in our new network?

Weights from Input to Hidden = Input Nodes × Hidden Nodes = 3 × 12 = 36
Biases for Hidden Layer = Hidden Nodes = 12
Weights from Hidden to Output = Hidden Nodes × Output Nodes = 12 × 3 = 36
Biases for Output Layer = Output Nodes = 3
Total Parameters = (36 + 12) + (36 + 3) = 48 + 39 = 87 Parameters



In problems 2 & 3, we will explore classifying the MNIST dataset using a logistic classifier, and then a two-layer neural network.

Problem 2: Logistic Regression

In this problem, you will be building a linear classifier (specifically, a logistic regression model) on the MNIST data set (which we also saw in Homework 1). As a reminder, this is an image classification dataset, where each image is a hand-written digit. Take a look at Homework 1 to remind yourself what this dataset looks like.

Problem 2.0: Setting up the Data

First, we'll load our dataset and split it into a training set and a testing set. Here you are given code that does this for you, and you only need to run it.

We will use the scikit-learn class `StandardScaler` to standardize both the training and testing features. Notice that we **only** fit the `StandardScaler` on the training data, and *not* the testing data.

In [20]:
```
# Load the features and labels for the MNIST dataset
# This might take a minute to download the images.
X, y = fetch_openml('mnist_784', as_frame=False, return_X_y=True)

# Convert labels to integer data type
y = y.astype(int)
```

In [21]:
```
X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.1, random_state=seed, shuffle=True)

scaler = StandardScaler()
scaler.fit(X_tr)
X_tr = scaler.transform(X_tr)     # We can forget about the original values & work
X_te = scaler.transform(X_te)     #  just with the transformed values from here
```

Problem 2.1: Initial Training (10 points)

For this part of the problem, you will train on **just** the first 10000 training data points, and compute the training and test error rates.

- Be sure to set the random seed with `random_state=seed` for consistency.
- Other than the random seed, just use the default values of the learner for this part.
- Here, the training error rate is defined on the first 10k data points (i.e., the points that were used for training the model
- The test error rate is defined on the full test data from your split.

In [22]:
```
m_tr = 10000

X_tr_subset = X_tr[:m_tr, :]
y_tr_subset = y_tr[:m_tr]

### YOUR CODE STARTS HERE ###

# Construct a logistic regression classifier (random_state = seed)
LR_model = LogisticRegression(random_state=seed)

# Fit the model to the training data subset
LR_model.fit(X_tr_subset, y_tr_subset)
```

```
# Compute the training error (on the subset) and testing error (on the test data)
y_tr_pred = LR_model.predict(X_tr_subset)
y_te_pred = LR_model.predict(X_te)

train_error = zero_one_loss(y_tr_subset, y_tr_pred)
test_error = zero_one_loss(y_te, y_te_pred)

print(f"Training Error: {train_error}")
print(f"Test Error: {test_error}")

### YOUR CODE ENDS HERE ###
```

```
Training Error: 0.0013999999999999568
Test Error: 0.12042857142857144
```

Problem 2.2: Regularization (10 points)

Suspecting that we are overfitting to our limited data set, we decide to try to use regularization. (This should reduce our model's variance, and thus its tendency to overfit.) Try re-training your logistic regression model at various levels of regularization.

The `LogisticRegression` class in `sklearn` takes an "inverse regularization" parameter, `C` (effectively the same as the value R we saw in soft-margin Support Vector Machines). Re-train your model with values of $C \in \{.0001, .001, .01, .1, 1.0, 10.\}$ and compute the training and test error rates of each setting. Plot the training and test error rates together as a function of C (use `semilogx`) and state what value of C you would select and why.

In [23]:
```
m_tr = 10000
C_vals = [.0001,.001,.01,.1,1.,10.];

### YOUR CODE STARTS HERE ###
train_errors = []
test_errors = []

# Train a logistic regression model with each inverse regularization C

for C_val in C_vals:
    LR_model2 = LogisticRegression(C=C_val, random_state=seed)

    # Compute the training and test error rates
    LR_model2.fit(X_tr_subset, y_tr_subset)

    y_tr_pred2 = LR_model2.predict(X_tr_subset)
    y_te_pred2 = LR_model2.predict(X_te)

    train_error = zero_one_loss(y_tr_subset, y_tr_pred2)
    test_error = zero_one_loss(y_te, y_te_pred2)

    # Append errors to the lists
    train_errors.append(train_error)
    test_errors.append(test_error)

# Plot the resulting performance as a function of C
plt.figure(figsize=(10, 6))
plt.semilogx(C_vals, train_errors, label='Training error')
plt.semilogx(C_vals, test_errors, label='Test error')
plt.xlabel('Inverse Regularization Strength (C)')
plt.ylabel('Error Rate')
plt.title('Training and Test Error Rates vs. Regularization Strength')
plt.legend()
plt.show()

### YOUR CODE ENDS HERE ###
```

Based on the plot above, we would select **.01** as the value of C becasue it results in the lowest test error without a significant difference from the training error, which would indicate a well-generalizing model.

Problem 2.3: Interpreting the weights (5 points)

Now that we have a model that we believe might perform well, let's try to understand what propertes of the data it is using to make its predictions. Since our model is just using a linear combination of the input pixels, we can display the coefficient (slope) associated with each pixel, to see whether that pixel's being bright (high value) is positively associated with a given class, or is negatively associated with that class.

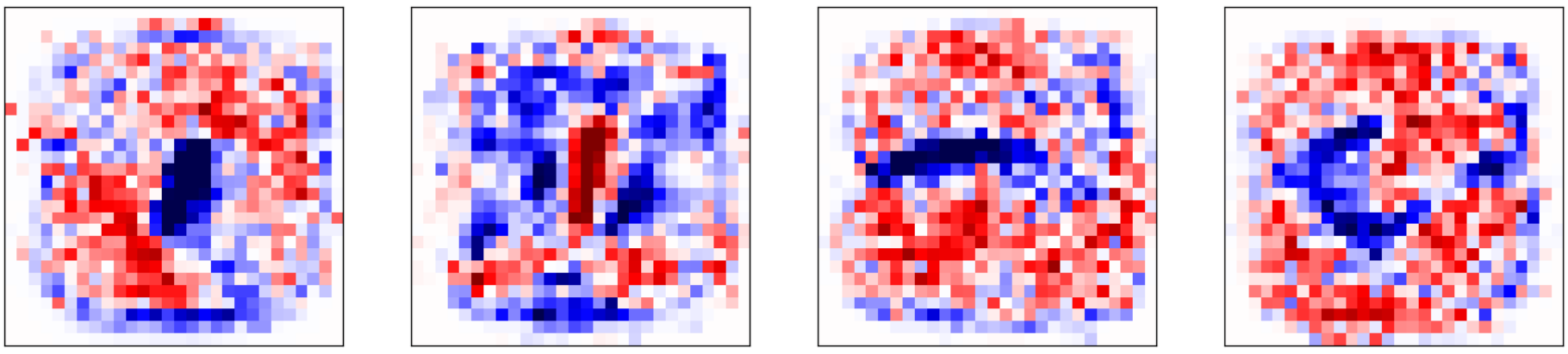
First, re-train your model using your selected value of C.

In [24]:
```
### YOUR CODE START HERE ###

# Re-train your model with your selected value of C
logistic_regression = LogisticRegression(C=.01, random_state=seed)
logistic_regression.fit(X_tr, y_tr)

### YOUR CODE ENDS HERE ###
```

Out[24]:
```
LogisticRegression
LogisticRegression(C=0.01, random_state=1234)
```

Run the provided code to display the coefficients of the first four classes' linear responses, re-shaped to the same size as the input image. (Here, red is positive, blue is negative, and white is zero.) Do the responses make sense? Discuss.

In [25]:
```
fig, ax = plt.subplots(1,4, figsize=(18,8))

mu = logistic_regression.coef_.mean(0).reshape(28,28)
for i in range(4):
    ax[i].imshow(logistic_regression.coef_[i,:].reshape(28,28)-mu,cmap='seismic',vmin=-.2,vmax=.2);
    ax[i].set_xticks([]); ax[i].set_yticks([]);
```



DISCUSS

Red areas: These correspond to positive weights, which means that higher pixel values (lighter areas in the actual image) in these regions are important for predicting the respective class.
Blue areas: These correspond to negative weights, indicating that lower pixel values (darker areas in the image) in these regions are important for predicting the respective class.
White areas: These are weights close to zero and do not contribute significantly to the prediction for the respective class.

The responses (plots) above do make sense. As we can see in the first plot, the red parts indicate the pen strokes in that area (positive). We can see that it is a circle which represents correct strokes for the digit 0. For the same plot, there is a very dense blue part in the center that shows there hasn't been any pen strokes there (negative). Similarly for the rest of the plots (responses), the red part shows the pen strokes (positive), making it look like a digit, the blue part is where there is no pen stroke (negative), and the white parts are just around the corners and on the sides which aren't of importance to us.

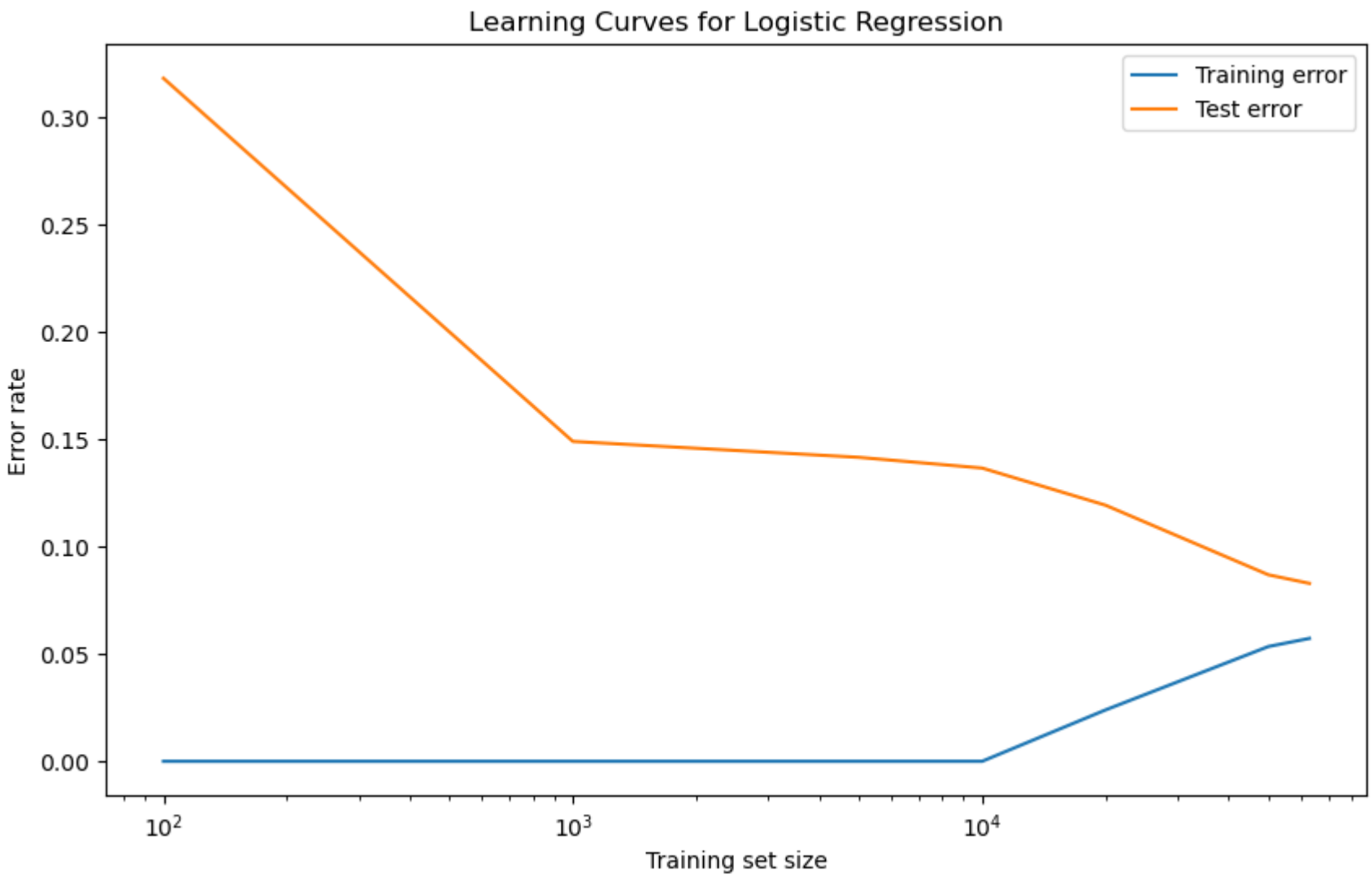
Problem 2.4: Learning Curves (10 points)

Another way to reduce overfitting is to increase the amount of data used for training the model (if possible). Build a logistic regression model, but with no regularization

- Train a logistic regression classifier (with the default settings in sklearn) using the first `m_tr` feature vectors in `X_tr`, where `m_tr = [100, 1000, 5000, 10000, 20000, 50000, 63000]` .You should use the `LogisticRegression` class from scikit-learn in your implementation. **Make sure to use the argument `random_state=seed` for reproducibility.**
- Create a plot of the training error and testing error for your logistic regression model as a function of the number of training data points. Be sure to include an x-label, y-label, and legend in your plot. Use a log-scale on the x-axis. Give a short (one or two sentences) description of what you see in your plot.
- Add a comment with your thoughts after the plot: although we ran out of data at 63k examples, can you tell how much additional data could help, with this model?

(Note: be sure to save these values, as we will want to plot them again in a later part of the homework.)

In [26]:
```
train_sizes = [100, 1000, 5000, 10000, 20000, 50000, 63000]

C = np.inf      # No regularization!

### YOUR CODE STARTS HERE ###
train_errors = []
test_errors = []

# Train a logistic regression model with each data size m and C=infinity

for size in train_sizes:
    LR_model3 = LogisticRegression(C=C, random_state=seed)
    LR_model3.fit(X_tr[:size], y_tr[:size])

    # Predict on the training subset and the entire test set
    y_tr_pred3 = LR_model3.predict(X_tr[:size])
    y_te_pred3 = LR_model3.predict(X_te)

    # Compute the training and test error rates
    train_errors.append(zero_one_loss(y_tr[:size], y_tr_pred3))
    test_errors.append(zero_one_loss(y_te, y_te_pred3))

# Plot the resulting performance as a function of m

plt.figure(figsize=(10, 6))
plt.plot(train_sizes, train_errors, label='Training error')
plt.plot(train_sizes, test_errors, label='Test error')
plt.xscale('log')
plt.xlabel('Training set size')
plt.ylabel('Error rate')
plt.title('Learning Curves for Logistic Regression')
plt.legend()
# plt.grid(True)
plt.show()

### YOUR CODE ENDS HERE ###
```



The learning curve plot above shows that the test error decreases with increasing training set size, indicating improved model generalization, while the training error remains at zero until 10,000 examples, suggesting overfitting to small datasets. Beyond 10,000 examples, the training error begins to rise, reflecting a transition towards a better fitting model as it encounters more diverse data.

The model could still benefit from additional data, and further improvements in generalization to new data might be achieved with a larger dataset. However, it's challenging to predict exactly how much additional data would be beneficial. The rate of improvement seems to be slowing, so while more data is expected to help, the gains may become increasingly marginal.

Problem 3: Neural Networks in Code

In this part of the assignment, you will get some hands-on experience working with neural networks. We will be using the scikit-learn implementation of a multi-layer perceptron (MLP). See here for the corresponding documentation. Although there are specialized Python libraries for neural networks, like TensorFlow and PyTorch, we'll stick with scikit-learn since you're already familiar with this library.

We will be working with the same MNIST data as in Problem 2, and comparing our neural network's performance to that of logistic regression.

Problem 3.1: Varying the amount of training data (15 points)

One reason that neural networks have become popular in recent years is that, for many problems, we now have access to very large datasets. Since neural networks are very flexible models, they are often able to take advantage of these large datasets in order to achieve high levels of accuracy. In this problem, you will vary the amount of training data available to a neural network and see what effect this has on the model's performance.

In this problem, you should use the following settings for your network:

- A single hidden layer with 64 hidden nodes
- Use the ReLU activation function
- Train the network using stochastic gradient descent (SGD) and a constant learning rate of 0.001
- Use a batch size of 256
- **Make sure to set `random_state=seed`.**

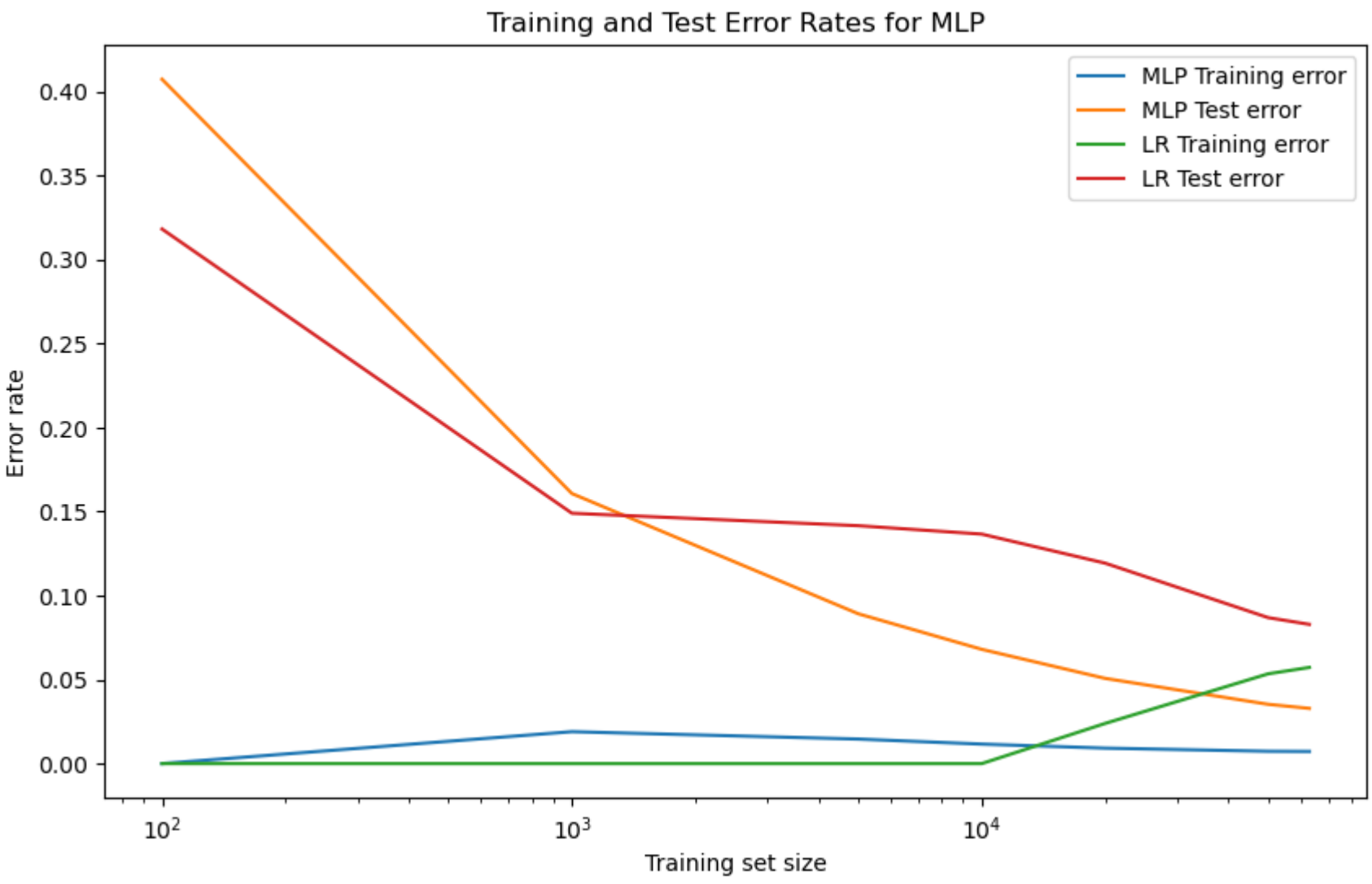
Your task is to implement the following:

- Train an MLP model (with the above hyperparameter settings) using the first `m_tr` feature vectors in `X_tr`, where `m_tr = [100, 1000, 5000, 10000, 20000, 50000, 63000]`. You should use the `MLPClassifier` class from scikit-learn in your implementation.

- Create a plot of the training error and testing error for your MLP model as a function of the number of training data points. For comparison, also plot the training and test error rates you found for logistic regression in Problem 2. Again, be sure to include an x-label, y-label, and legend in your plot and use a log-scale on the x-axis.
- Give a short (one or two sentences) description of what you see in your plot. Do you think that more data (beyond these 63000 examples) would continue to improve the model's performance?

Note that training a neural network with a lot of data can be **a slow process**. Hence, you should be careful to implement your code such that it runs in a reasonable amount of time. One recommendation is to test your code using only a small subset of the given `m_tr` values, and only run your code with the larger values of `m_tr` once you are certain your code is working. (For reference, it took about 20 minutes to train all models on a quad-core desktop with no GPU.)

In [21]:
```
train_sizes = [100, 1000, 5000, 10000, 20000, 50000, 63000]

### YOUR CODE STARTS HERE ###
mlp_train_errors = []
mlp_test_errors = []



# Train your neural network model with each data size m and given hyperparameter values

for m in train_sizes:
    mlp_model = MLPClassifier(hidden_layer_sizes=(64,), activation='relu', solver='sgd',
                        batch_size=256, learning_rate_init=0.001, random_state=seed)
    mlp_model.fit(X_tr[:m], y_tr[:m])

    # Compute the training and test error rates
    mlp_train_errors.append(zero_one_loss(y_tr[:m], mlp_model.predict(X_tr[:m])))
    mlp_test_errors.append(zero_one_loss(y_te, mlp_model.predict(X_te)))

# Plot the resulting performance as a function of m

plt.figure(figsize=(10, 6))
plt.plot(train_sizes, mlp_train_errors, label='MLP Training error')
plt.plot(train_sizes, mlp_test_errors, label='MLP Test error')
plt.plot(train_sizes, train_errors, label='LR Training error')
plt.plot(train_sizes, test_errors, label='LR Test error')
plt.xscale('log')
plt.xlabel('Training set size')
plt.ylabel('Error rate')
plt.title('Training and Test Error Rates for MLP')
plt.legend()
# plt.grid(True)
plt.show()

### YOUR CODE ENDS HERE ###
```



The convergence of training and test error rates as the training set size approaches 63,000 suggests that the model is becoming more consistent in its predictions, with a diminishing gap indicating less overfitting. Given this trend, additional data could likely continue to marginally improve the model's performance, particularly in enhancing its generalization capabilities.

Problem 3.2: Optimization Curves (10 points)

One hyperparameter that can have a significant effect on the optimization of your model, and thus its performance, is the learning rate, which controls the step size in (stochastic) gradient descent. In this problem you will vary the learning rate to see what effect this has on how quickly training converges as well as the effect on the performance of your model.

In this problem, you should use the following settings for your network:

- A single hidden layer with 64 hidden nodes
- Use the ReLU activation function
- Train the network using stochastic gradient descent (SGD)
- Use a batch size of 256
- Set `n_iter_no_change=100` and `max_iter=100`. This ensures that all of your networks in this problem will train for 100 epochs (an *epoch* is one full pass over the training data).
- Make sure to set `random_state=seed`.

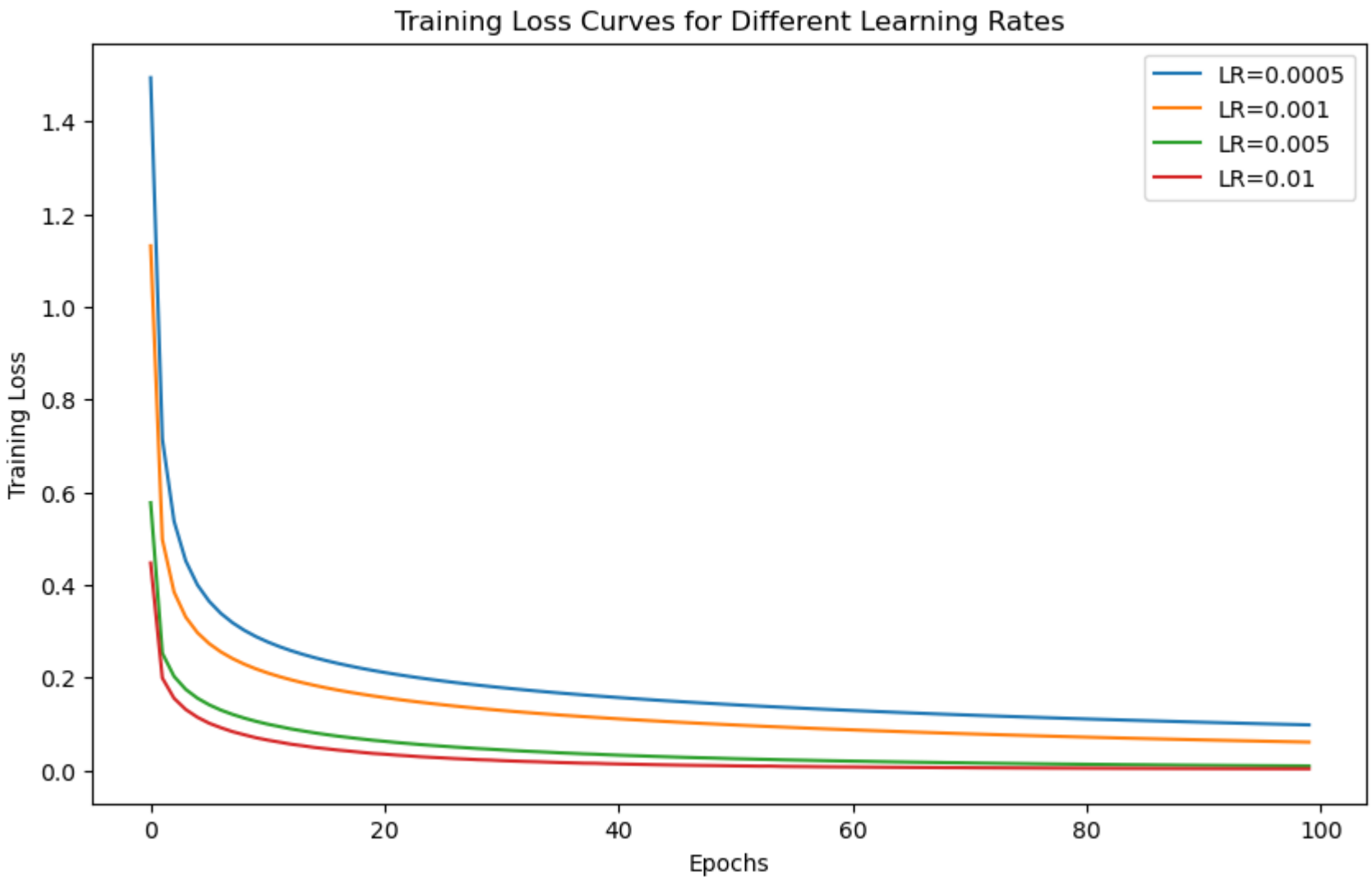
Your task is to:

- Train a neural network with the above settings, but vary the learning rate in `lr = [0.0005, 0.001, 0.005, 0.01]`.
- Create a plot showing the training loss as a function of the training epoch (i.e. the x-axis corresponds to training iterations) for each learning rate above. You should have a single plot with four curves. Make sure to include an x-label, a y-label, and a legend in your plot. (Hint: `MLPClassifier` has an attribute `loss_curve_` that you likely find useful.)
- Include a short description of what you see in your plot.

Important: To make your code run faster, you should train all of your networks in this problem on only the first 10,000 images of `X_tr`. In the following cell, you are provided a few lines of code that will create a small training set (with the first 10,000 images in `X_tr`) and a validation set (with the second 10,000 images in `X_tr`). You will use the validation later in Problem 3.3.

In [7]:
```
# Create a smaller training set with the first 10,000 images in X_tr
#   along with a validation set from images 10,000 - 20,000 in X_tr

X_val = X_tr[10000:20000] # Validation set
y_val = y_tr[10000:20000]

X_tr = X_tr[:10000]       # From here on, we will only use these smaller sets,
y_tr = y_tr[:10000]       #  so it's OK to discard the rest of the data
```

In [22]:
```
learning_rates = [0.0005, 0.001, 0.005, 0.01]

### YOUR CODE STARTS HERE ###
loss_curves = {}

# Train your neural network model on the small data using each learning rate

for lr in learning_rates:
    mlp_model2 = MLPClassifier(hidden_layer_sizes=(64,), activation='relu', solver='sgd',
                        batch_size=256, learning_rate_init=lr, max_iter=100,
                        n_iter_no_change=100, random_state=seed, verbose=False)
    mlp_model2.fit(X_tr, y_tr)

    # Store the loss curve
    loss_curves[lr] = mlp_model2.loss_curve_

# Compute the training and test error rates for each model

# Plot the training loss curve for each setting (single plot) & compare

plt.figure(figsize=(10, 6))
for lr, loss_curve in loss_curves.items():
    plt.plot(loss_curve, label=f'LR={lr}')

plt.xlabel('Epochs')
plt.ylabel('Training Loss')
plt.title('Training Loss Curves for Different Learning Rates')
plt.legend()
# plt.grid(True)
plt.show()

### YOUR CODE ENDS HERE ###
```



Problem 3.3: Tuning a Neural Network (10 points)

As you saw in Problem 3.2, there are many hyperparameters of a neural network that can possibly be tuned in order to try to maximize the accuracy of your model. For the final problem of this assignment, it is your job to tune these hyperparameters.

For example, some hyperparameters you might choose to tune are:

- Learning rate
- Depth/width of the hidden layers
- Regularization strength
- Activation functions
- Batch size in stochastic optimization
- etc.

To do this, you should train a network on the training data `X_tr` and evaluate its performance on the validation set `X_val` -- your goal is to achieve the highest possible accuracy on `X_val` by changing the network hyperparameters. **Important: To make your code run faster, you should train all of your networks in this problem on only the first 10,000 images of `X_tr`**. This was already set up for you in Problem 3.2.

Try to find settings that enable you to achieve an error rate smaller than 5% on the validation data. However, tuning neural networks can be difficult; if you cannot achieve this target error rate, be sure to try at least five different neural networks (corresponding to five different settings of the hyperparameters).

In your answer, include a table listing the different hyperparameters that you tried, along with the resulting accuracy on the training and validation sets `X_tr` and `X_val`. Indicate which of these hyperparameter settings you would choose for your final model, and report the accuracy of this final model on the testing set `X_te`.

In [17]:
```
import pandas as pd
from sklearn.model_selection import ParameterSampler

# X_val = X_tr[10000:20000] # Validation set
# y_val = y_tr[10000:20000]

# X_tr = X_tr[:10000] # From here on, we will only use these smaller sets,
# y_tr = y_tr[:10000]

# Define the hyperparameters to try
hyperparameters_list = [
    {'learning_rate_init': 0.001, 'hidden_layer_sizes': (64,), 'alpha': 0.0001},
    {'learning_rate_init': 0.001, 'hidden_layer_sizes': (128,), 'alpha': 0.0001},
    {'learning_rate_init': 0.001, 'hidden_layer_sizes': (64, 64), 'alpha': 0.0001},
    {'learning_rate_init': 0.01, 'hidden_layer_sizes': (64,), 'alpha': 0.001},
    {'learning_rate_init': 0.01, 'hidden_layer_sizes': (128,), 'alpha': 0.001},
]

results = []

for hyperparams in hyperparameters_list:
```

```
    # train the MLPClassifier
    mlp = MLPClassifier(
        hidden_layer_sizes=hyperparams['hidden_layer_sizes'],
        activation='relu',
        solver='sgd',
        batch_size=256,
        learning_rate_init=hyperparams['learning_rate_init'],
        alpha=hyperparams['alpha'],
        max_iter=200,
        n_iter_no_change=10,
        random_state=seed,
        verbose=False
    )
    mlp.fit(X_tr, y_tr)

    # Predict and calculate the accuracy on the training set
    y_pred_train = mlp.predict(X_tr)
    train_accuracy = accuracy_score(y_tr, y_pred_train)

    # Predict and calculate the accuracy on the validation set
    y_pred_val = mlp.predict(X_val)
    val_accuracy = accuracy_score(y_val, y_pred_val)

    # Append the results
    results.append({
        'learning_rate_init': hyperparams['learning_rate_init'],
        'hidden_layer_sizes': hyperparams['hidden_layer_sizes'],
        'alpha': hyperparams['alpha'],
        'train_accuracy': train_accuracy,
        'val_accuracy': val_accuracy,
        'model': mlp
    })

# Convert the results to a DataFrame
results_df = pd.DataFrame(results)

# Sort by validation accuracy
results_df.sort_values('val_accuracy', ascending=False, inplace=True)

# Display the table
print(results_df[['learning_rate_init', 'hidden_layer_sizes', 'alpha', 'train_accuracy', 'val_accuracy']])

# Select the best model based on validation accuracy
best_model_info = results_df.iloc[0]
best_model = best_model_info['model']

# Predict and calculate the accuracy on the test set
y_pred_test = best_model.predict(X_te)
test_accuracy = accuracy_score(y_te, y_pred_test)

# Report the accuracy of the final model on the test set
print(f"Final model's accuracy on test set: {test_accuracy:.2f}")
```

```
   learning_rate_init hidden_layer_sizes   alpha  train_accuracy  val_accuracy
4               0.010             (128,)  0.0010          1.0000        0.9447
3               0.010              (64,)  0.0010          1.0000        0.9419
1               0.001             (128,)  0.0001          0.9907        0.9371
2               0.001           (64, 64)  0.0001          0.9944        0.9347
0               0.001              (64,)  0.0001          0.9884        0.9341
Final model's accuracy on test set: 0.94
```

In [27]:

```
#    learning_rate_init hidden_layer_sizes   alpha  train_accuracy  val_accuracy
# 4               0.010             (128,)  0.0010          1.0000        0.9447
# 3               0.010              (64,)  0.0010          1.0000        0.9419
# 1               0.001             (128,)  0.0001          0.9907        0.9371
# 2               0.001           (64, 64)  0.0001          0.9944        0.9347
# 0               0.001              (64,)  0.0001          0.9884        0.9341
```

The best model (highest validation accuracy) that I got is the one with the following hyperparameters,

- hidden_layer_sizes: 128
- activation: 'relu'
- solver: 'sgd',
- batch_size: 256,
- learning_rate_init: 0.010
- alpha: 0.0010
- max_iter: 200
- n_iter_no_change: 10
- random_state: 1234
- verbose: False

Accuracy of the final model: 0.9447. Therefore, error rate = 1 - 0.9447 = 0.0553 (5.53%)

Statement of Collaboration (5 points)

It is **mandatory** to include a Statement of Collaboration in each submission, with respect to the guidelines below. Include the names of everyone involved in the discussions (especially in-person ones), and what was discussed.

All students are required to follow the academic honesty guidelines posted on the course website. For programming assignments, in particular, I encourage the students to organize (perhaps using EdD) to discuss the task descriptions, requirements, bugs in my code, and the relevant technical content before they start working on it. However, you should not discuss the specific solutions, and, as a guiding principle, you are not allowed to take anything written or drawn away from these discussions (i.e. no photographs of the blackboard, written notes, referring to EdD, etc.). Especially after you have started working on the assignment, try to restrict the discussion to EdD as much as possible, so that there is no doubt as to the extent of your collaboration.

I did not work with anyone on this Homework. I just used Ed and professor office hours to clear doubts.