cs273-hw3-Abdul_Kalam_Syed

October 27, 2025

1 CS273 Homework 3

1.1 Due: Friday October 24 2025 (11:59pm)

1.2 Instructions

Like Homework 1, this homework involves 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.

Again, you may use this Jupyter notebook as a template to get you started. 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.

1.2.1 Summary of Assignment: 100 total points

- Problem 1: Logistic Regression (25 points)
  - Problem 1.1: Decision boundaries (10 points)
  - Problem 1.2: Gradient optimization (10 points)
  - Problem 1.3: Evaluation (5 points)
- Problem 2: Linear Support Vector Machines (15 points)
  - Problem 2.1: Fitting & Evaluation (8 points)
  - Problem 2.2: Decision boundary & margin (7 points)
- Problem 3: Feature Expansions (20 points)
  - Problem 3.1: Polynomial Features (10 points)
  - Problem 3.2: Using Regularization (10 points)
- Problem 4: Logistic Regression on MNIST Data (35 points)

  - Problem 4.1: Initial Training (10 points)
  - Problem 4.2: Regularization (10 points)
  - Problem 4.3: Interpreting the Weights (5 points)
  - Problem 4.4: Evaluating class probabilities (5 points)
  - Problem 4.4: Learning Curves (5 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.

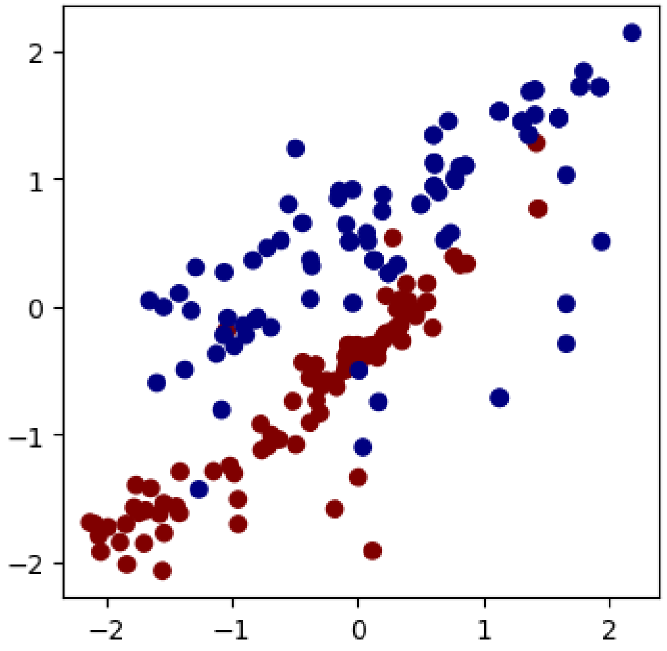
```
[2]: import numpy as np
     import matplotlib.pyplot as plt

     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 zero_one_loss
     from sklearn.inspection import DecisionBoundaryDisplay

     from sklearn.preprocessing import PolynomialFeatures
     from sklearn.preprocessing import StandardScaler

     from sklearn.linear_model import SGDClassifier        # Used in 2D data problems
     from sklearn.linear_model import LogisticRegression   # Used in MNIST data␣
      ↪problem


     import requests             # we'll use these for reading data from a url
     from io import StringIO

     import warnings
     warnings.filterwarnings('ignore')

     # Some keyword arguments for making nice looking decision plots.
     plot_kwargs = {'cmap': 'jet',      # another option: viridis
                    'response_method': 'predict',
                    'plot_method': 'pcolormesh',
                    'shading': 'auto',
                    'alpha': 0.5,
                    'grid_resolution': 100}
```

```
# Fix the random seed for reproducibility
# !! Important !! : do not change this
seed = 1234
np.random.seed(seed)
```

1.3 Binary Classification Dataset

First, let's load our Housing dataset from HW1. To start, we will extract a two-dimensional binary classification problem, which will allow us to visualize the problem, training, and resulting model.

[3]:
```
# Load the features and labels from an online text file
url = 'https://ics.uci.edu/~ihler/classes/cs178/data/nyc_housing.txt'
with requests.get(url) as link:
    datafile = StringIO(link.text)
    nych = np.genfromtxt(datafile,delimiter=',')
    nych_X, nych_y = nych[:,:-1], nych[:,-1]

# Process the data to be only two classes and two real-valued & normalized␣
 ↪features:
X, y = nych_X[nych_y<2,:2],nych_y[nych_y<2]
X -= X.mean(axis=0,keepdims=True)  # remove mean
X /= X.std(axis=0,keepdims=True)   # & scale
y = 2*y - 1                        # classical binary: positive/negative

# Visualize the resulting dataset:
plt.figure(figsize=(4,4))
plt.scatter(X[:,0],X[:,1],c=y,cmap='jet');
```



1.4 Problem 1: Logistic Regression

The `scikit` package contains several implementations of logistic regression models for classification. In order to emphasize the similarities between different models, we will use the `SGDClassifier` object, which is a bit of a misnomer since SGD is an optimization technique, not a model. The object implements several types of linear classifiers, optimized using SGD or SGD-like training, depending on the loss function selected.

1.4.1 Problem 1.1: Decision Boundaries

First, let's build a linear classifier and manually set its parameters. Suppose that we initialize our linear classifier to make it's predictions as,

$$\hat{y} = T(\ \theta_0 + \theta_1 x_1 + \theta_2 x_2\ )$$

with $[\ \theta_0,\ \theta_1,\ \theta_2\ ] = [-2,\ 2,\ 1\ ]$.

(a) What is the decision boundary of this classifier? (Answer in the form $x_2 = a x_1 + b$.)

Answer: $x_2 = -2x_1 + 2$

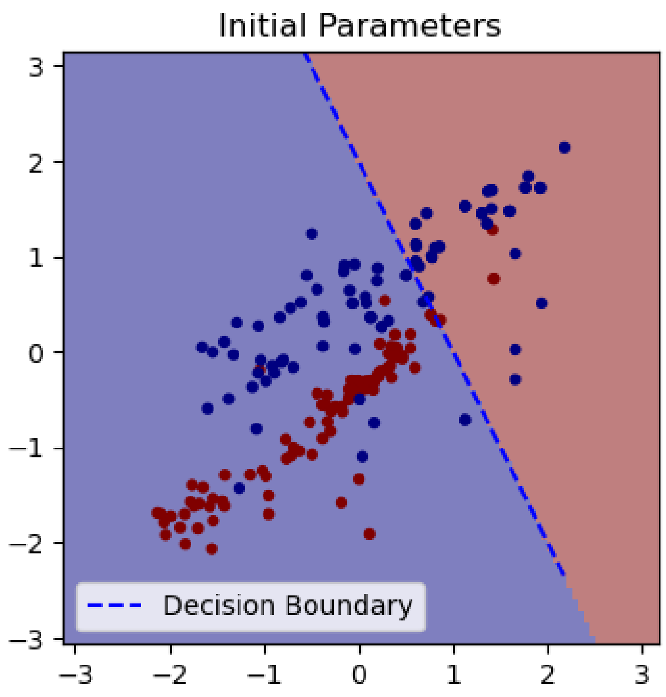
Let's initialize the classifier and look at its decision function.

We will set the classifer to use the logistic negative log-likelihood surrogate loss (`loss=log_loss`); the other parameters prevent re-initializing the model later (`warm_start=True`) and set the stochastic gradient step size schedule (`learning_rate='adaptive'` is a simple backoff method, and initial step size `eta0=1e-3` is a small initial step size, so we can see the early progress).

(b) Add code below to plot your answer above on the decision function and verify that your answer matches `scikit`'s output:

[4]:
```
logreg = SGDClassifier(loss='log_loss', warm_start=True,␣
 ↪learning_rate='adaptive', eta0 = 1e-3)

# Now let's initialize the model manually:
logreg.classes_ = np.unique(y)       # class IDs from the data
logreg.coef_ = np.array([[2.,1.]])
logreg.intercept_ = np.array([-2.])   # r(x) = 2*x1 + 1*x2 + (-2)

figure, axes = plt.subplots(1, 1, figsize=(4,4))
DecisionBoundaryDisplay.from_estimator(logreg, X, ax=axes, **plot_kwargs)
axes.scatter(X[:, 0], X[:, 1], c=y, edgecolor=None, s=12, cmap='jet')

### YOUR CODE STARTS HERE

# Plot the line you derived in part 1 on the figure, in an appropriate range of␣
 ↪values
x_vals = np.linspace(X[:,0].min(), X[:,0].max(), 100) # x1 values
y_vals = 2 - 2 * x_vals  # derived from 2*x1 + 1*x2 - 2 = 0  =>  x2 =  -2*x1 + 2
```

```
axes.plot(x_vals, y_vals, color='blue', linestyle='--', label='Decision␣
↪Boundary')
axes.legend()

# Setting the limits of the plot to ensure all data points are visible
axes.set_xlim(X[:,0].min() - 1, X[:,0].max() + 1)
axes.set_ylim(X[:,1].min() - 1, X[:,1].max() + 1)


### YOUR CODE ENDS HERE

axes.set_title(f'Initial Parameters');
```



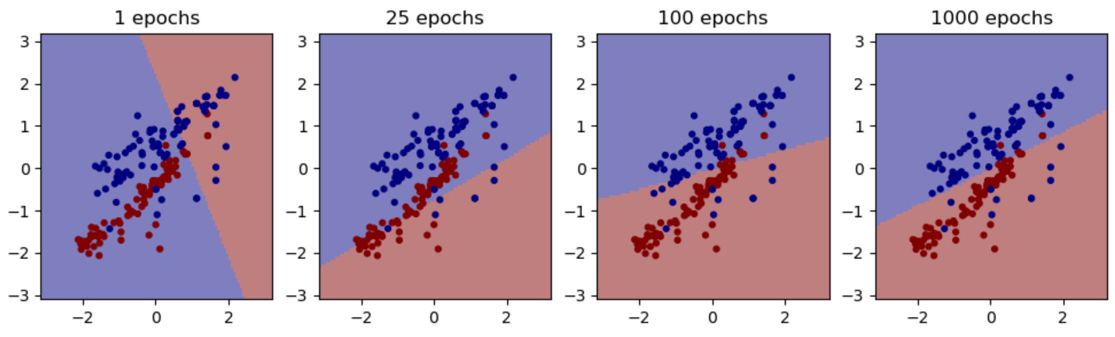
1.4.2 Problem 1.2: Gradient Optimization

Start training your model using stochastic gradient descent, and looking at the classifier and decision boundary as you progress. For this part, we use `partial_fit`, a function that does a single epoch of stochastic gradient descent, and does not reset the internal state of the optimization loop (number of iterations, etc.), so that subsequent calls "pick up" right where the previous calls left off.

We'll initialize the model as before; then, train your model and visualize its current decision function (using `DecisionBoundaryDisplay`) after each of: * 1 epoch * 25 epochs * 100 epochs * 1000 epochs (final model)

Note that each call to `partial_fit` performs one epoch of SGD.

[23]:
```
np.random.seed(seed)

logreg = SGDClassifier(loss='log_loss', warm_start=True,␣
 ↪learning_rate='adaptive', eta0 = 1e-3)
logreg.coef_ = np.array([[2.,1.]])
logreg.intercept_ = np.array([-2.])   # r(x) = 2*x1 + 1*x2 + (-2)

plot_iters = [1,25,100,1000]
figure, axes = plt.subplots(1, 4, figsize=(12,3))

### YOUR CODE STARTS HERE
# Using partial fits of the logistic regression model

epochs_done = 0 # to keep track of how many epochs have been completed to not␣
 ↪train more than needed

for iters, ax in zip(plot_iters, axes):
    # run just the additional epochs needed to reach `iters`
    for j in range(iters - epochs_done): # since we are using partial_fit, we␣
 ↪only need to do the difference of already done epochs
        logreg.partial_fit(X, y, classes=np.unique(y))

    epochs_done = iters

    # ploting current decision boundary
    DecisionBoundaryDisplay.from_estimator(logreg, X, ax=ax, **plot_kwargs)
    ax.scatter(X[:, 0], X[:, 1], c=y, s=12, cmap='jet')
    ax.set_title(f'{iters} epochs')


### YOUR CODE ENDS HERE
```



1.4.3 Problem 1.3: Evaluation

Using your final model after training, display its learned linear coefficients and evaluate its (training) error rate.

Manually compute the linear response at the point $(x_1, x_2) = (-1, 0)$, and use the logistic function to evaluate the model's estimated probability that this point is in each class. (You can use the model's built-in function `predict_proba` to check your answer, if you like.)

[40]:
```
# Display model parameters
print("Model coefficients (weights):", logreg.coef_)
print("Model intercept (bias):", logreg.intercept_)

# Evaluate model performance
y_pred = logreg.predict(X) # predicted labels
train_error = np.mean(y_pred != y) # training error rate
print(f"Training Error Rate: {train_error}")

# Manual computation: linear response and predicted probability at (-1,0)
point = np.array([-1, 0]) # point we want to evaluate
lr = logreg.intercept_[0] + np.dot(logreg.coef_[0], point) # linear response
p = 1 / (1 + np.exp(-lr)) # predicted probability
print(f"Linear response r(x): {lr}")
print(f"Predicted probability for class +1: {p:.2f}")
```

```
Model coefficients (weights): [[ 1.50940898 -3.47550824]]
Model intercept (bias): [-0.02326915]
Training Error Rate: 0.095
Linear response r(x): -1.532678125122346
Predicted probability for class +1: 0.18
```

[41]:
```
logreg.predict_proba([[-1,0]]).round(2)  # Evaluate on final model to check␣
  ↪your answer
```

[41]:
```
array([[0.82, 0.18]])
```

1.5 Problem 2: (Linear) Support Vector Machines

As we saw in lecture, a linear support vector machine optimizes the "margin" around the data. Our current data set is not linearly separable, so we will need to use a "Soft Margin" SVM. Soft-margin Linear SVMs are equivalent to a linear classifier trained using an L2-regularized hinge loss; so, we can implement the SVM using exactly the same `SGDClassifier` model, using the same learner (linear classifier) and an identical learning algorithm (stochastic gradient), but changing the loss function.

To make our model as "close" to a hard-margin SVM as possible, we set the L2 regularization to be very small. This also can make the optimization a bit slow, so we'll use a lot of iterations and turn off any early stopping criteria.

1.5.1 Problem 2.1: Training & Evaluation

Fit your model to the data, then print out its linear coefficients and the resulting (training) error rate:

[42]:
```
np.random.seed(seed)

learner = SGDClassifier(loss='hinge',             # hinge loss = primal linear␣
 ↪SVM form
              penalty='l2',alpha=1e-20,           # small L2 regularization is␣
 ↪"closest" to Hard SVM
              learning_rate='adaptive',eta0=1e-3, # same optmization as before
              tol=0.,max_iter=10000,n_iter_no_change=1000)  # prevent any early␣
 ↪stopping

### YOUR CODE STARTS HERE

# Train the model, display your parameters & evaluate its performance
learner.fit(X, y) # train the model

# Print model parameters
print("Model coefficients (weights):", learner.coef_)
print("Model intercept (bias):", learner.intercept_)

# Evaluate model performance
y_pred = learner.predict(X) # predicted labels
train_error = np.mean(y_pred != y) # training error rate
print(f"Training Error Rate: {train_error}")

### YOUR CODE ENDS HERE
```

```
Model coefficients (weights): [[ 1.72112436 -2.63600577]]
Model intercept (bias): [0.14569216]
Training Error Rate: 0.075
```

1.5.2 Problem 2.2: Decision boundary & margins

Now, display the decision function learned by your linear SVM. In addition, on top of the decision boundary plot, display the SVM's margins, i.e.,

$$r(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 = +1$$

and

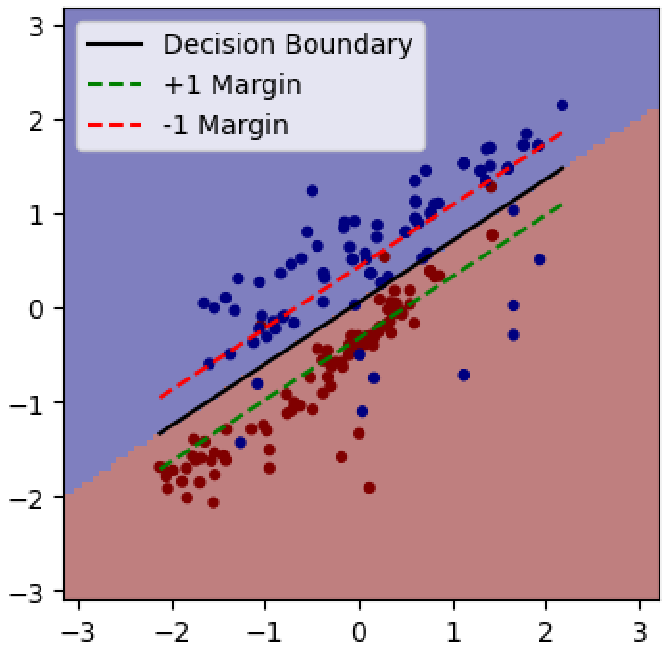
$$r(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 = -1$$

(Recall that the decision boundary, as you plotted earlier, is given by $r(x) = 0$.)

[43]:
```
figure, axes = plt.subplots(1, 1, figsize=(4,4))
DecisionBoundaryDisplay.from_estimator(learner, X, ax=axes, **plot_kwargs)
axes.scatter(X[:, 0], X[:, 1], c=y, edgecolor=None, s=12, cmap='jet')
```

```
### YOUR CODE STARTS HERE

# Draw (e.g. with dashed lines) the set of points that are on the +1 and -1␣
↪margins

# (Hint: this is almost the same as drawing the decision boundary earlier,␣
↪except that
#  you need to use your trained parameters, and solve r(x)=+1 and r(x)=-1␣
↪instead of r(x)=0.)

# Get the trained parameters
weights = learner.coef_[0]
bias = learner.intercept_[0]

# Calculate the margin lines
x1_vals = np.linspace(X[:,0].min(), X[:,0].max(), 100)

# Getting the decision boundary
decision_boundary = (-bias - weights[0]*x1_vals) / weights[1]

# For r(x) = 1: weights[0]*x1 + weights[1]*x2 + bias = 1  =>  x2 = (1 - bias -␣
↪weights[0]*x1) / weights[1]
margin_pos = (1 - bias - weights[0]*x1_vals) / weights[1]
# For r(x) = -1: weights[0]*x1 + weights[1]*x2 + bias = -1  =>  x2 = (-1 - bias␣
↪- weights[0]*x1) / weights[1]
margin_neg = (-1 - bias - weights[0]*x1_vals) / weights[1]

# Plot the decision boundary
axes.plot(x1_vals, decision_boundary, color='black', linestyle='-',␣
↪label='Decision Boundary')
# Plot the margin lines
axes.plot(x1_vals, margin_pos, color='green', linestyle='--', label='+1 Margin')
axes.plot(x1_vals, margin_neg, color='red', linestyle='--', label='-1 Margin')
axes.legend()

### YOUR CODE ENDS HERE
```

```
[43]: <matplotlib.legend.Legend at 0x15ab2135d50>
```



1.6 Problem 3: Feature Expansion

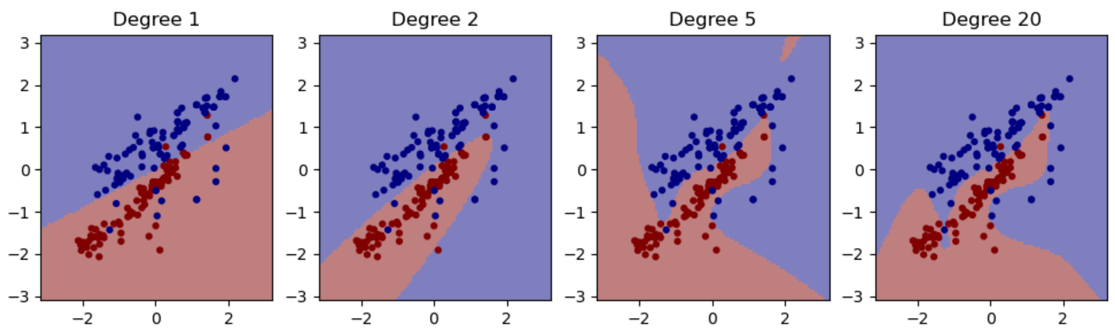
If we feel that our linear classifier is insufficiently flexible, one option is to provide it with more features. Just like in our linear regression models, additional features, such as polynomial features, make the resulting model more adaptable to the data.

In this problem, we will expand our features using `PolynomialFeatures`, and look at the resulting logistic regression model's decision function.

Note that, when creating new features, especially high-order polynomials, it is a good idea to scale the data after the feature transform. As in the HW2 solutions, the easiest way to expand the feature set and rescale the data is to use the `Pipeline` object in `sklearn`.

Adapt the code below to fit and display the decision function for degrees 1, 2, 5, and 20.

[46]:
```
from sklearn.pipeline import Pipeline
np.random.seed(seed)

degrees=[1,2,5,20]
figure,axes = plt.subplots(1,4,figsize=(12,3))

for i,d in enumerate(degrees):

# Each item in the pipeline is a pair, (name, transform); the end is (name,␣
↪learner):
    learner = Pipeline( [('poly',PolynomialFeatures(degree=d)),
                         ('scale',StandardScaler()),
                          ('logreg',SGDClassifier(loss='log_loss',
                                                  penalty='l2',alpha=1e-20,
                                                  learning_rate='adaptive',␣
↪eta0=1e-2,
                                                  tol=0.
↪,max_iter=100000,n_iter_no_change=1000))
                         ])

    ### YOUR CODE STARTS HERE

    # Fit the model
    learner.fit(X, y)

    # Display the resulting decision function and training data
    DecisionBoundaryDisplay.from_estimator(learner, X, ax=axes[i],␣
↪**plot_kwargs)
    axes[i].scatter(X[:, 0], X[:, 1], c=y, s=12, cmap='jet')
    axes[i].set_title(f'Degree {d}')

    ### YOUR CODE ENDS HERE
```



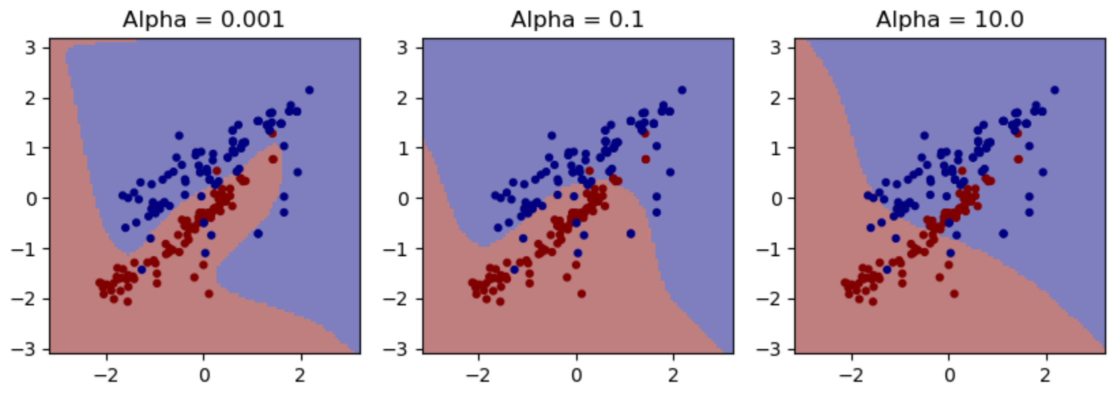
1.6.1 Problem 3.2: Regularization

Our higher-order models are most likely overfitting (although we can't tell for sure, since we didn't save any data for validation). Let's re-learn the model using some regularization to see how it affects the resulting decision function.

Try increasing the L2 regularization to `1e-3`, `1e-1`, and `10` and display the resulting decision functions. Discuss how these compare to each other, and to the (nearly) unregularized version in the previous question.

[48]:
```
from sklearn.pipeline import Pipeline
np.random.seed(seed)

d = 20
```

```
alphas = [1e-3, 1e-1, 10.]
figure,axes = plt.subplots(1,3,figsize=(10,3))

for i,alpha in enumerate(alphas):

# Each item in the pipeline is a pair, (name, transform); the end is (name,␣
↪learner):
    learner = Pipeline( [('poly',PolynomialFeatures(degree=d)),
                         ('scale',StandardScaler()),
                         ('logreg',SGDClassifier(loss='log_loss',
                                                 penalty='l2',alpha=alpha,
                                                 learning_rate='adaptive',␣
↪eta0=1e-2,
                                                 tol=0.
↪,max_iter=100000,n_iter_no_change=1000))
                        ])

    ### YOUR CODE STARTS HERE

    # Fit the model
    learner.fit(X, y)

    # Display the resulting decision function and training data
    DecisionBoundaryDisplay.from_estimator(learner, X, ax=axes[i],␣
↪**plot_kwargs)
    axes[i].scatter(X[:, 0], X[:, 1], c=y, s=12, cmap='jet')
    axes[i].set_title(f'Alpha = {alpha}')

    ### YOUR CODE ENDS HERE
```



Discussion When we have too low regularization, the model tends to overfit the training data by trying to include most of the data points making the decision boundary very complex. When we increase the regularization strength too much, the model over simplifies the decision boundary, leading to underfitting as it ignores clear patterns in the data. With moderate regularization, the model achieves a good balance between fitting the training data and maintaining a simpler decision boundary that generalizes better to unseen data. The same things can be observed with different polynomial degrees in the prevous question.

1.7 Problem 4: Logistic Regression on MNIST

Finally, let us now build a linear classifier (specifically, a logistic regression model) on a higher-dimensional, multi-class problem: the MNIST data set.

The MNIST dataset is an image dataset consisting of 70,000 hand-written digits (from 0 to 9), each of which is a 28x28 grayscale image. For each image, we also have a label, corresponding to which digit is written.

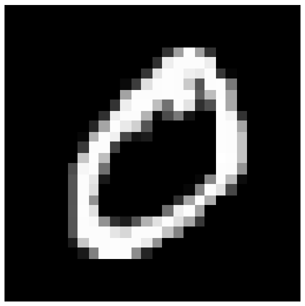
1.7.1 Problem 4.0: Setting up the Data

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

[49]:
```
# 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)
```

Each data point in the MNIST dataset is 784-dimensional, with each feature corresponding to a pixel intensity of a 28×28 scan of a digit. To visualize a data point, we can re-shape the feature vector into the shape of the image, and then display it using `imshow`:

[50]:
```
plt.figure(figsize=(2,2))
plt.imshow( X[1,:].reshape(28,28) , cmap='gray');
plt.axis('off');
```



As before, we will normalize the data before learning using 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.

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

      X_tr_orig, X_te_orig = X_tr, X_te   # Save a copy of these for later␣
       ↪visualization

      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
                                        # (This does make it harder to visualize a␣
       ↪data point, though)
```

1.7.2 Problem 4.1: 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.

```
[52]: m_tr = 10000

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

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

      ### YOUR CODE STARTS HERE ###

      # Fit your model to the (small subset of the) training data
      learner_mnist.fit(X_tr_subset, y_tr_subset)

      # Compute the training error (on the small training subset)
      train_pred = learner_mnist.predict(X_tr_subset)
      train_error = np.mean(train_pred != y_tr_subset)
      print(f"Training Error Rate: {train_error}")

      # testing error (on the test data)
      test_pred = learner_mnist.predict(X_te)
      test_error = np.mean(test_pred != y_te)
      print(f"Testing Error Rate: {test_error}")
```

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

```
Training Error Rate: 0.0014
Testing Error Rate: 0.12042857142857143
```

Your model should learn a set of linear coefficients for each of the 10 classes:

[53]:
```
print(f'Coefficients shape: {learner_mnist.coef_.shape}')    # should be 10 x␣
↪784
print(f'Intercepts shape: {learner_mnist.intercept_.shape}') # should be 10
```

```
Coefficients shape: (10, 784)
Intercepts shape: (10,)
```

1.7.3 Problem 4.2: Regularization (10 points)

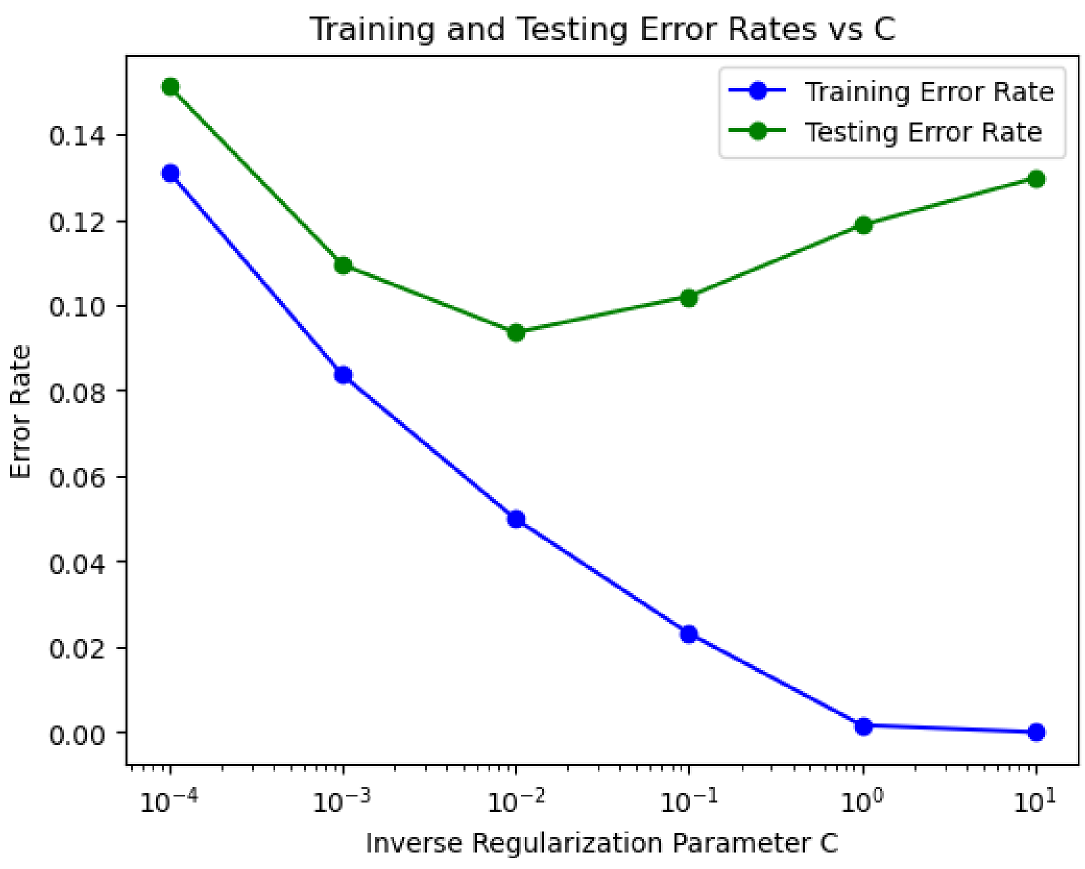
Suspecting that we are overfitting to our limited data set, we decide to try to use regularization. (This should reduce our model's variance, and thus its tendency to overfit.) Try re-training your logistic regression model at various levels of regularization.

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 (plot using `semilogx` for it to look nice) and state what value of C you would select and why.

[56]:
```
m_tr = 10000
C_vals = [.0001,.001,.01,.1,1.,10.];

### YOUR CODE STARTS HERE ###
train_error = []
test_error = []

for C in C_vals:

    # Train a logistic regression model with each inverse regularization C
    learner_mnist = LogisticRegression(C=C, random_state=seed, max_iter=1000)
    learner_mnist.fit(X_tr_subset, y_tr_subset)

    # Computing the training error rate
    train_pred = learner_mnist.predict(X_tr_subset)
    train_error_rate = np.mean(train_pred != y_tr_subset)
    train_error.append(train_error_rate)

    # Computing the testing error rate
    test_pred = learner_mnist.predict(X_te)
    test_error_rate = np.mean(test_pred != y_te)
    test_error.append(test_error_rate)
    print(f"C={C}: Training Error Rate: {train_error_rate}, Testing Error Rate:␣
↪{test_error_rate}")

# Plot the resulting performance as a function of C
plt.semilogx(C_vals, train_error, label='Training Error Rate', color='blue',␣
↪marker='o')
plt.semilogx(C_vals, test_error, label='Testing Error Rate', color='green',␣
↪marker='o')
plt.xlabel('Inverse Regularization Parameter C')
plt.ylabel('Error Rate')
plt.title('Training and Testing Error Rates vs C')
plt.legend()

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

```
C=0.0001: Training Error Rate: 0.1311, Testing Error Rate: 0.15114285714285713
C=0.001: Training Error Rate: 0.0836, Testing Error Rate: 0.10942857142857143
C=0.01: Training Error Rate: 0.0499, Testing Error Rate: 0.09357142857142857
C=0.1: Training Error Rate: 0.0231, Testing Error Rate: 0.102
C=1.0: Training Error Rate: 0.0016, Testing Error Rate: 0.11871428571428572
C=10.0: Training Error Rate: 0.0, Testing Error Rate: 0.12971428571428573
```

```
[56]: <matplotlib.legend.Legend at 0x15ab2c2ec90>
```



I would select C = 0.1 because the testing error is the lowest at that point and the training error is also low. It would be a balanced choice without overfitting or underfitting.

1.7.4 Problem 4.3: 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.

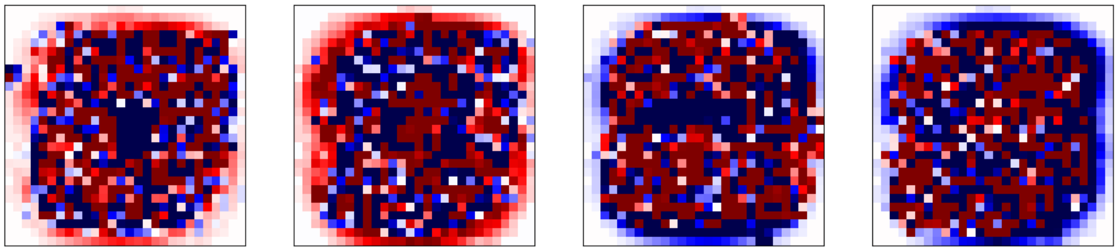
First, re-train your model using your selected value of C.

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

# Re-train your model with your selected value of C
selected_C = 0.1  # I would select C = 0.1 because the testing error is the␣
 ↪lowest at that point and the training error is also low. It would be a␣
 ↪balanced choice without overfitting or underfitting.
learner_mnist = LogisticRegression(C=selected_C, random_state=seed,␣
 ↪max_iter=1000)
```

```
learner_mnist.fit(X_tr_subset, y_tr_subset)

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

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.

[57]:
```
fig, ax = plt.subplots(1,4, figsize=(18,8))

mu = learner_mnist.coef_.mean(0).reshape(28,28)
for i in range(4):
    ax[i].imshow(learner_mnist.coef_[i,:].
 ↪reshape(28,28)-mu,cmap='seismic',vmin=-.25,vmax=.25);
    ax[i].set_xticks([]); ax[i].set_yticks([]);
```



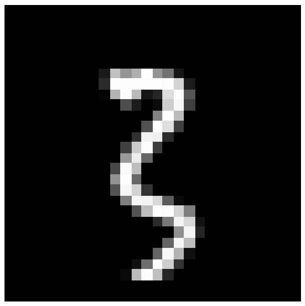
[ ]:
```
### DISCUSS
# The responses do make sense. Given that the coefficient vectors represent the␣
 ↪weights assigned to each pixel for classifying digits, we can view them as␣
 ↪highlighting the most important features for each digit.
# Though the plots appear noisy, we can still see certain patterns that␣
 ↪correspond to the shapes of the digits 0, 1, 2, and 3.
# Since red indicates positive weights and blue indicates negative weights, we␣
 ↪can see that for each digit we can see that the color patterns roughly align␣
 ↪with the strokes that form each digit.
```

1.7.5 Problem 4.4

The multilogistic classifier uses the negative log-likelihood loss, just like the logistic classifier, but produces a predicted probability for each class based on that class's linear response.

In this problem, we'll consider a particular (somewhat ambiguous) data point:

[58]:
```
idx = 10592
plt.figure(figsize=(2,2))
plt.imshow( X_tr_orig[idx,:].reshape(28,28) , cmap='gray');  # Use "orig" to␣
 ↪show it before rescaling
plt.axis('off');
```



(a) Using your model parameters, **manually** compute the linear response values for each of the 10 classes on this data point. (You can do this easily using matrix multiplication and addition of arrays.)

```
[ ]: point = X_tr[idx, :]
     linear_responses = learner_mnist.intercept_ + np.dot(learner_mnist.coef_, point)
     print("Linear responses for each class:", linear_responses)
```

```
Linear responses for each class: [ -3.25758287  14.75452081  -2.40151412
8.34778801 -22.77608948
   3.46268196  -3.96761935   1.83734426   7.9637572   -3.96328641]
```

(b) Use the multi-logit or `softmax` transformation to convert these responses into estimated class probabilities.

```
[60]: #**(b)** Use the ``softmax`` transformation to convert these responses into␣
        ↪estimated class probabilities.
      estimated_probabilities = np.exp(linear_responses) / np.sum(np.
        ↪exp(linear_responses))
      print("Estimated class probabilities:", estimated_probabilities)
```

```
Estimated class probabilities: [1.50048958e-08 9.97218247e-01 3.53198553e-08
1.64581689e-03
 5.00556687e-17 1.24396142e-05 7.37680080e-09 2.44867824e-06
 1.12098307e-03 7.40883341e-09]
```

(c) Do these probabilities make sense, given the observation? Discuss briefly.

```
[ ]: # The probilities do make sense given the estimated class highest probability␣
       ↪corresponds to the correct class label for this data point.
     # Though the probabilities for some other classes are not negligible, it still␣
       ↪make sense beacuse it finds the image somewhat similar to those classes as␣
       ↪well.
```

Note: To check your answer, you can compare to the values given by the learner's built-in `predict_proba()` function:

```
[61]: learner_mnist.predict_proba(X[idx:idx+1,:]).round(2)
```

```
[61]: array([[0., 0., 0., 0., 0., 0., 0., 0., 1., 0.]])
```

1.7.6 Problem 4.5: Learning Curves (10 points)

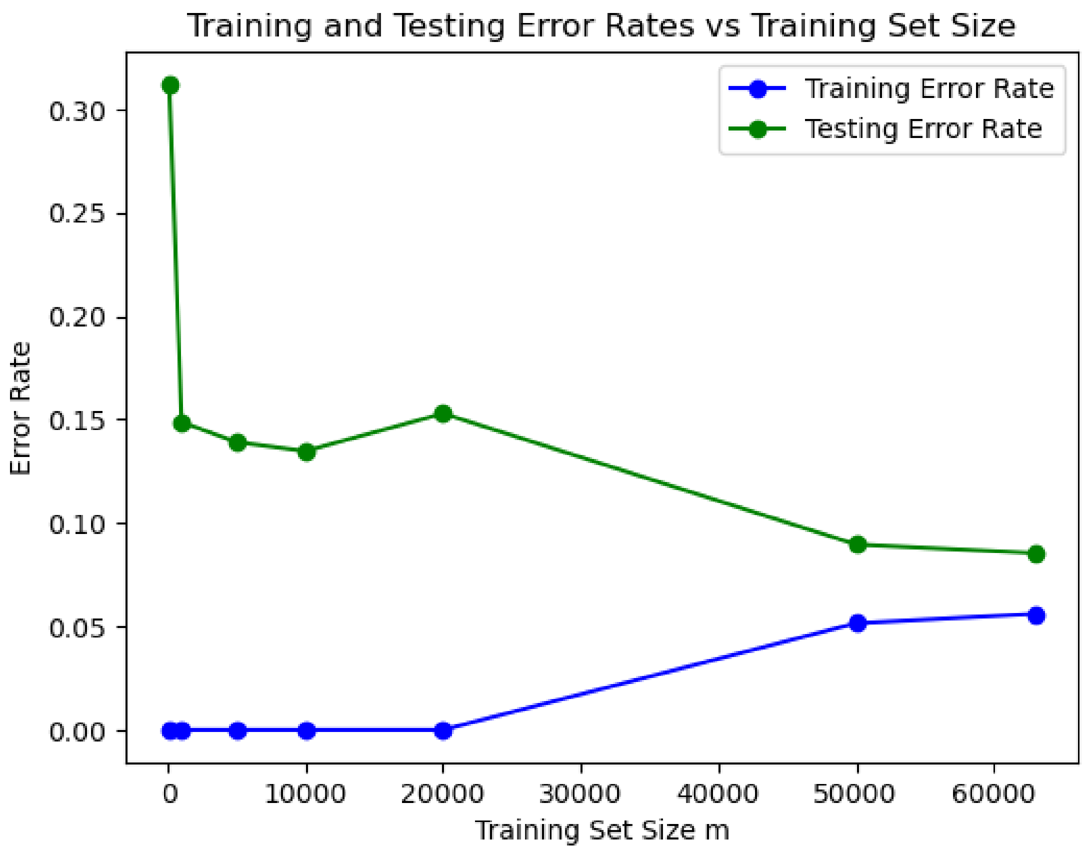
Another way to reduce overfitting is to increase the amount of data used for training the model (if possible). Build a logistic regression model, but with no regularization

- 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?

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

      C = np.inf      # No regularization!

      ### YOUR CODE STARTS HERE ###

      # Train a logistic regression model with each data size m and C=infinity
      train_error = []
      test_error = []

      for m in train_sizes:
          X_tr_subset = X_tr[:m, :]
          y_tr_subset = y_tr[:m]

          # Traninig model with logistic regression with no regularization
          learner_mnist = LogisticRegression(C=C, random_state=seed, max_iter=1000)
          learner_mnist.fit(X_tr_subset, y_tr_subset)

          # Compute the training error rate
          train_pred = learner_mnist.predict(X_tr_subset)
          train_error_rate = np.mean(train_pred != y_tr_subset)
          train_error.append(train_error_rate)

          # Compute test error rate
          test_pred = learner_mnist.predict(X_te)
          test_error_rate = np.mean(test_pred != y_te)
          test_error.append(test_error_rate)
```

```
# Plot the resulting performance as a function of m
plt.plot(train_sizes, train_error, label='Training Error Rate', color='blue',␣
  ↪marker='o')
plt.plot(train_sizes, test_error, label='Testing Error Rate', color='green',␣
  ↪marker='o')
plt.xlabel('Training Set Size m')
plt.ylabel('Error Rate')
plt.title('Training and Testing Error Rates vs Training Set Size')
plt.legend()

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

[64]:
```
<matplotlib.legend.Legend at 0x15ab40f8a50>
```



[ ]:
```
# COMMENT / DISCUSS
# We can see that the training error decreases as the training set size␣
  ↪increases, which makes sense because with more training data, the model has␣
  ↪more information to learn from and can better fit the training data.
```

```
# We can see the model is improving its generalization as the training set size␣
↪increases, as evidenced by the decreasing testing error.
# Since the testing error still decreasing, adding more training data would␣
↪likely continue to improve performance. May be another 10k data points could␣
↪help further.
```

```
<img src="data:image/svg+xml,%3C%3Fxml%20version%3D%221.0%22%20encoding%3D%22UTF-8%22%20standa
```

1.7.7 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 have not collaborated with anyone on this assignment.