cs273_hw3

October 21, 2024

1 CS273 Homework 3

1.1 Due: Friday October 25 2024 (11:59pm)

1.2 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: 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.

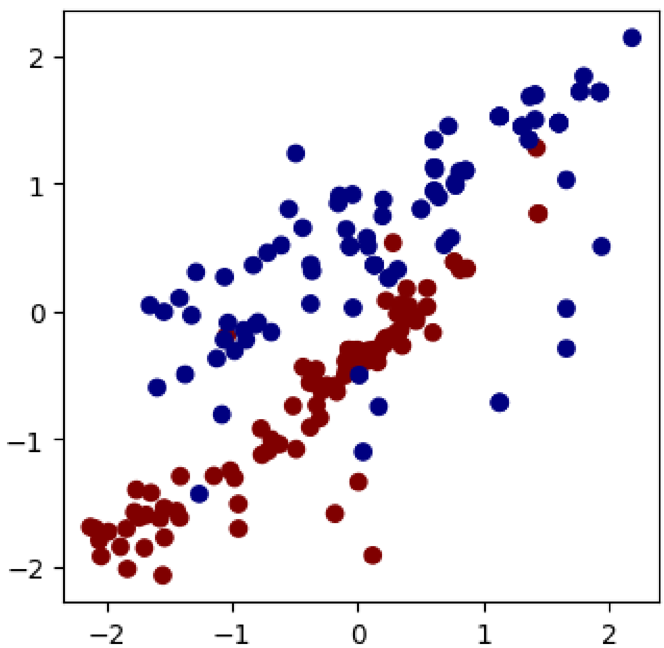
Important: Do not change any codes we give you below, except for those waiting for you to complete. This is to ensure your code has reproducible results and is important for grading.

[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]:
```python
# Load the features and labels from an online text file
url = 'https://sli.ics.uci.edu/extras/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 $x_2 = ax_1 + b$.)

x_2 = -2*x_1 +2

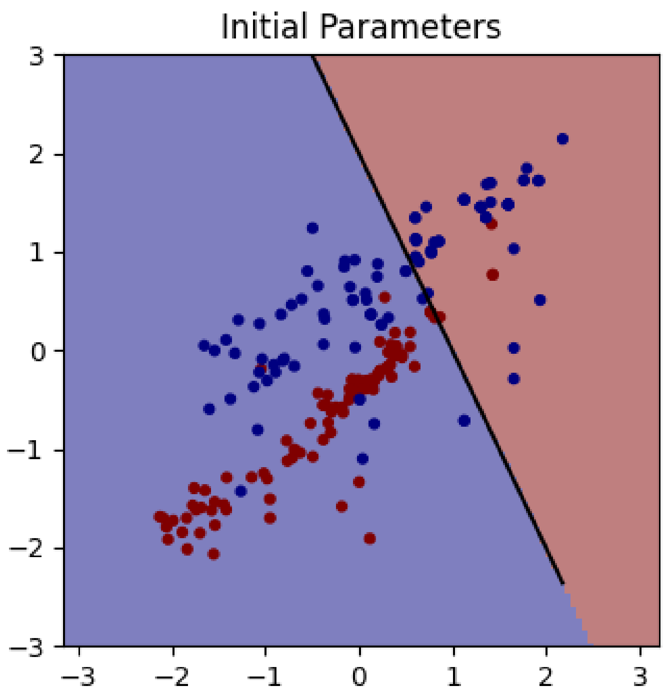
Let's initialize the classifier and look at its decision function.

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:

[9]:
```
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 * x_vals + 2  # Corresponding x2 values based on the line equation

# Plot the line on the axes
```

```
axes.plot(x_vals, y_vals, color='black', label=r'Decision boundary: $x_2 =␣
↪-2x_1 + 2$')
axes.set_ylim([-3, 3])

### YOUR CODE ENDS HERE

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



1.4.2 Problem 1.2: Gradient Optimization

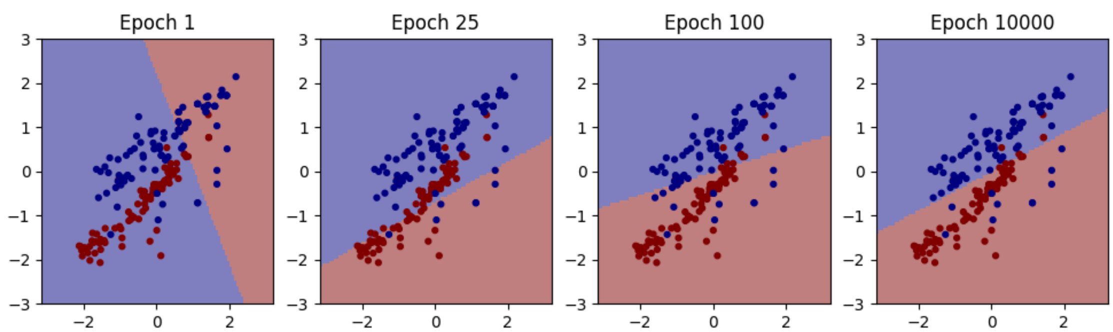
Start training your model using stochastic gradient descent, and looking at the classifier and decision boundary as you progress. For this part, we use `partial_fit`, a function that does a single epoch of stochastic gradient descent, and does not reset the internal state of the optimization loop (number 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.

[10]:
```
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 = [1,25,100,10000]
figure, axes = plt.subplots(1, 4, figsize=(12,3))

### YOUR CODE STARTS HERE
for idx, iters in enumerate(plot_iters):
    for _ in range(iters):
        logreg.partial_fit(X, y, classes=np.unique(y))

    DecisionBoundaryDisplay.from_estimator(logreg, X, ax=axes[idx],␣
↪**plot_kwargs)
    axes[idx].scatter(X[:, 0], X[:, 1], c=y, edgecolor=None, s=12, cmap='jet')
    axes[idx].set_title(f'Epoch {iters}')
    axes[idx].set_ylim([-3, 3])

### 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.)

[11]:
```
# Display model parameters
print(f"Learned coefficients (weights): {logreg.coef_}")
print(f"Learned intercept (bias term): {logreg.intercept_}")

# Evaluate model performance
y_pred = logreg.predict(X)
error_rate = zero_one_loss(y, y_pred)
```

```python
print(f"Training error rate: {error_rate * 100:.2f}%")

# Manual computation: linear response at (-1, 0)
x_manual = np.array([-1, 0])
linear_response = np.dot(logreg.coef_, x_manual) + logreg.intercept_
print(f"Linear response at (-1, 0): {linear_response}")

# Logistic function to compute probability: 1 / (1 + exp(-z))
probability_class_1 = 1 / (1 + np.exp(-linear_response))
probability_class_2 = 1 - probability_class_1
print(f"Probability of class 1: {probability_class_1[0]:.4f}")
print(f"Probability of class -1: {probability_class_2[0]:.4f}")

prob_model = logreg.predict_proba(x_manual.reshape(1, -1))
print(f"Model-predicted probabilities: {prob_model}")
```

```
Learned coefficients (weights): [[ 1.56416414 -3.54684729]]
Learned intercept (bias term): [-0.02672695]
Training error rate: 9.50%
Linear response at (-1, 0): [-1.59089109]
Probability of class 1: 0.1693
Probability of class -1: 0.8307
Model-predicted probabilities: [[0.83074144 0.16925856]]
```

[12]:
```python
logreg.predict_proba([[-1,0]]).round(2)  # Evaluate on final model to check␣
↪your answer
```

[12]:
```
array([[0.83, 0.17]])
```

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:

[13]:
```python
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)
print(f"Learned coefficients (weights): {learner.coef_}")
print(f"Learned intercept (bias term): {learner.intercept_}")

y_pred = learner.predict(X)
error_rate = zero_one_loss(y, y_pred)
print(f"Training error rate: {error_rate * 100:.2f}%")
### YOUR CODE ENDS HERE
```

```
Learned coefficients (weights): [[ 1.72112436 -2.63600577]]
Learned intercept (bias term): [0.14569216]
Training error rate: 7.50%
```

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.,

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

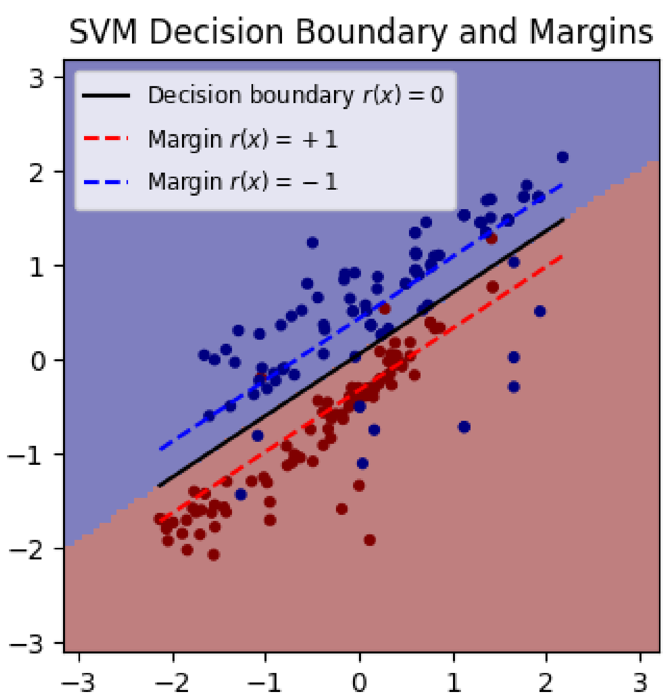
and

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

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

[17]:
```
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.)
```

```python
coef = learner.coef_[0]   # [1,  2]
intercept = learner.intercept_    #  0

x_vals = np.linspace(X[:, 0].min(), X[:, 0].max(), 100)

# Decision boundary: r(x) = 0 --> 0 + 1*x1 + 2*x2 = 0
decision_boundary = -(coef[0] * x_vals + intercept) / coef[1]

# Margin: r(x) = +1 and r(x) = -1
margin_plus_1 = -(coef[0] * x_vals + (intercept - 1)) / coef[1]  # r(x) = +1
margin_minus_1 = -(coef[0] * x_vals + (intercept + 1)) / coef[1]  # r(x) = -1

# Plot the decision boundary
axes.plot(x_vals, decision_boundary, 'k-', label=r'Decision boundary $r(x)=0$')

# Plot the margins with dashed lines
axes.plot(x_vals, margin_plus_1, 'r--', label=r'Margin $r(x)=+1$')
axes.plot(x_vals, margin_minus_1, 'b--', label=r'Margin $r(x)=-1$')

axes.set_title(f'SVM Decision Boundary and Margins')
axes.legend(loc='upper left', fontsize='small')
plt.show()
### YOUR CODE ENDS HERE
```



1.6 Problem 3: Feature Expansion

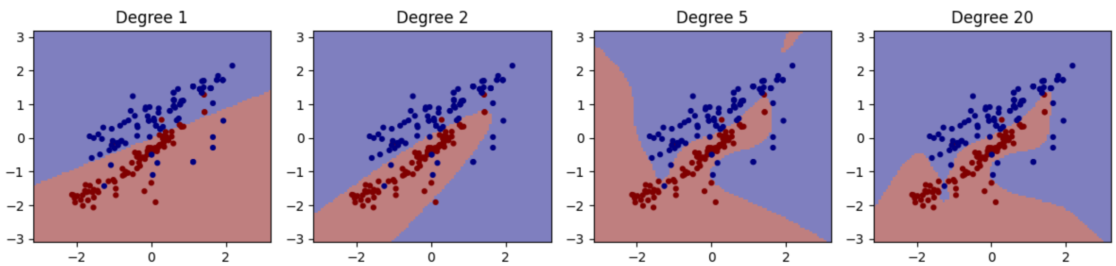
If we feel that our linear classifier is insufficiently flexible, one option is to provide it with more features. Just like in our linear regression models, additional features, such as polynomial features, make the resulting model more adaptable to the data.

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.

[18]:
```
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

    learner.fit(X, y)

    DecisionBoundaryDisplay.from_estimator(learner, X, ax=axes[i],␣
↪**plot_kwargs)
    axes[i].scatter(X[:, 0], X[:, 1], c=y, edgecolor=None, s=12, cmap='jet')

    axes[i].set_title(f'Degree {d}')

plt.tight_layout()
plt.show()
    ### YOUR CODE ENDS HERE
```



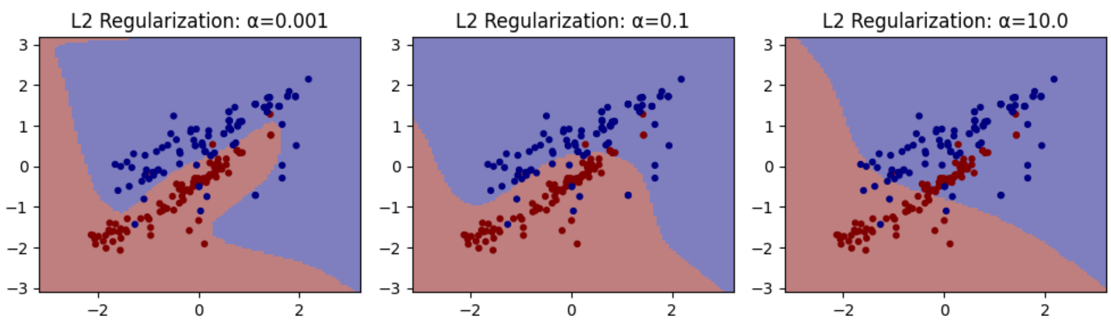
1.6.1 Problem 3.2: Regularization

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.

[19]:
```
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

    learner.fit(X, y)

    # Display the resulting decision function
```

```
    DecisionBoundaryDisplay.from_estimator(learner, X, ax=axes[i],␣
↪**plot_kwargs)
    axes[i].scatter(X[:, 0], X[:, 1], c=y, edgecolor=None, s=12, cmap='jet')

    # Set plot title to indicate the regularization strength (alpha)
    axes[i].set_title(f'L2 Regularization:  ={alpha}')

plt.tight_layout()
plt.show()
    ### YOUR CODE ENDS HERE
```



Lower Regularization (= 1e-3): The model likely overfits the training data, producing complex decision boundaries.

Moderate Regularization (= 1e-1): The model balances flexibility with smoothness, reducing overfitting but still adapting to the data.

High Regularization (= 10): The model significantly reduces overfitting, but may underfit, resulting in a smoother decision boundary.

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

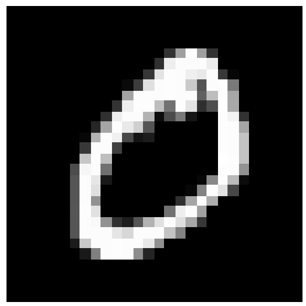
First, we'll load our dataset, split it into a training set and a testing set, and do some basic pre-processing. Here you are given code that does this for you, and you only need to run it.

[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)
```

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

Each data point in the MNIST dataset is 768-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`:

[21]:
```python
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.

[23]:
```python
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
                                   # (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.

[28]:
```
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) and testing error␣
↪(on the test data)
train_error = 1 - learner_mnist.score(X_tr_subset, y_tr_subset)
print(f"Training error: {train_error * 100:.2f}%")

test_error = 1 - learner_mnist.score(X_te, y_te)
print(f"Test error: {test_error * 100:.2f}%")
### YOUR CODE ENDS HERE ###
```

```
Training error: 0.14%
Test error: 12.04%
```

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

[29]:
```
print(f'Coefficients shape: {learner_mnist.coef_.shape}')    # should be 10 x␣
↪768
print(f'Intercepts shape: {learner_mnist.intercept_.shape}') # should be 10
```

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

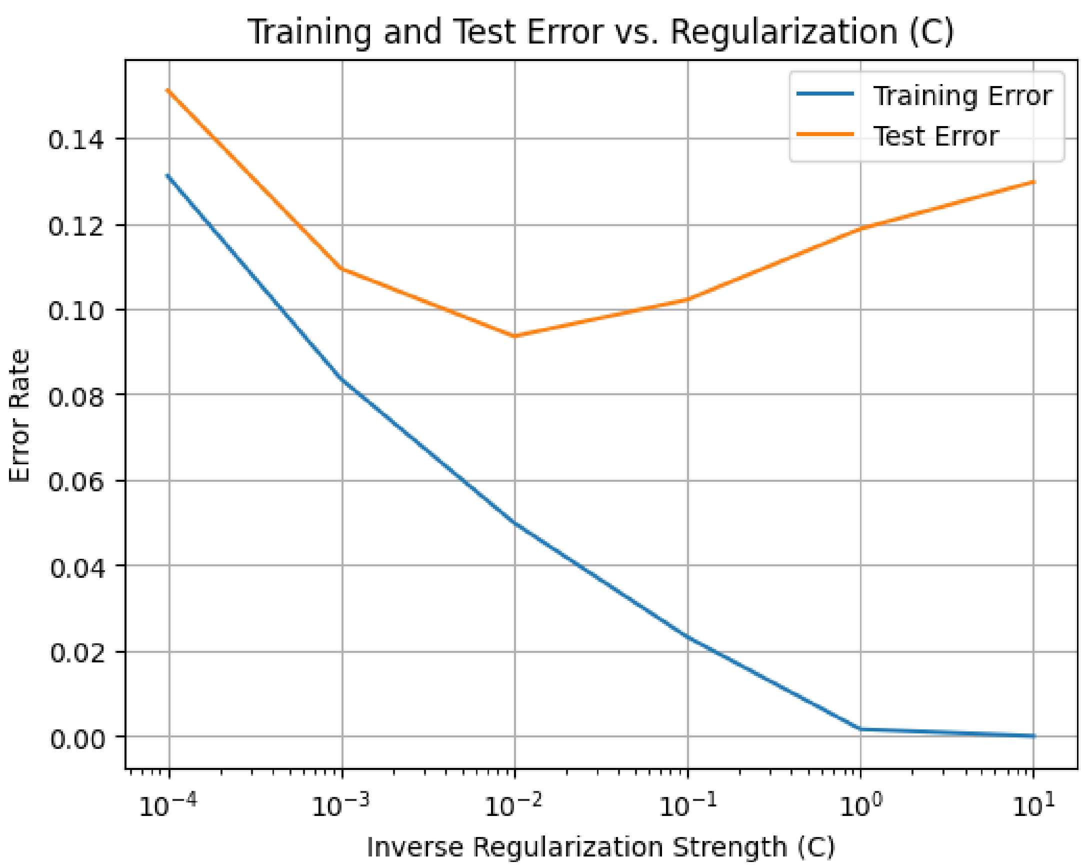
1.7.3 Problem 4.2: Regularization (10 points)

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.

[33]:
```
m_tr = 10000
C_vals = [.0001,.001,.01,.1,1.,10.];

### YOUR CODE STARTS HERE ###
```

```
training_errors = []
test_errors = []

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)
    # Compute the training and test error rates at each value of C
    training_errors.append(1 - learner_mnist.score(X_tr_subset, y_tr_subset))
    test_errors.append(1 - learner_mnist.score(X_te, y_te))
# Plot the resulting performance as a function of C
plt.semilogx(C_vals, training_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 vs. Regularization (C)')
plt.legend()
plt.grid(True)
plt.show()

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



[37]:
```python
# what value of c to consider
c = np.argmin(test_errors)
print(f"The value of C to consider is: {C_vals[c]}")
```

```
The value of C to consider is: 0.01
```

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.

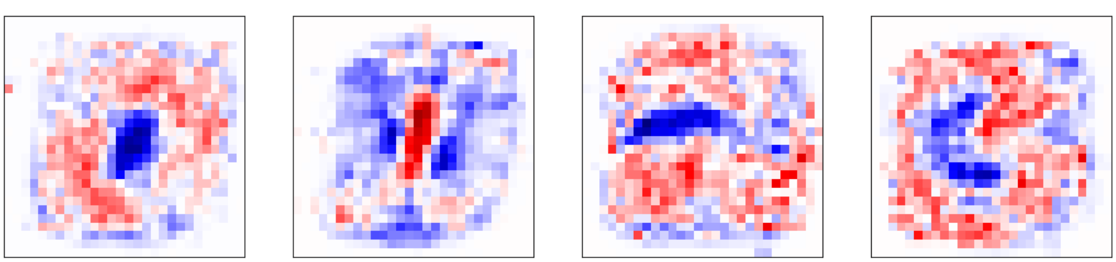
First, re-train your model using your selected value of C.

[40]:
```python
### YOUR CODE START HERE ###

# Re-train your model with your selected value of C
learner_mnist = LogisticRegression(C=C_vals[c], random_state=seed,␣
 ↪max_iter=1000)
learner_mnist.fit(X_tr_subset, y_tr_subset)

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

[40]:
```
LogisticRegression(C=0.01, max_iter=1000, 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.

[41]:
```python
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
```

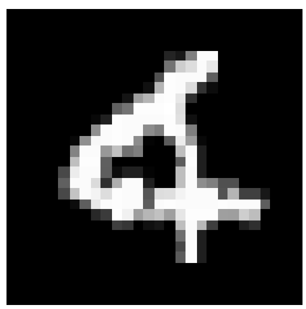
Yes the responses makes sense because if we look at the red portion, we see that they somewhat resembles the digits 0,1,2,3 which are the first four class.

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:

[ ]:
```
idx = 14290
plt.figure(figsize=(2,2))
plt.imshow( X[idx,:].reshape(28,28) , cmap='gray');
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.)

[44]:
```
x_idx = X[idx, :]

weights = learner_mnist.coef_
intercepts = learner_mnist.intercept_

linear_responses = np.dot(weights, x_idx) + intercepts

print(f"Linear responses for data point {idx}:")
for class_idx, response in enumerate(linear_responses):
    print(f"Class {class_idx}: {response:.4f}")
```

```
Linear responses for data point 3:
Class 0: -374.1543
```

```
Class 1: 541.5150
Class 2: 324.6417
Class 3: 70.4999
Class 4: -248.3183
Class 5: -335.2621
Class 6: -192.6415
Class 7: -179.1774
Class 8: 561.3908
Class 9: -168.4939
```

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

[45]:
```
# Compute the softmax transformation to get class probabilities
exp_responses = np.exp(linear_responses)
probabilities = exp_responses / np.sum(exp_responses)

# Display the probabilities for each class
print(f"Class probabilities for data point {idx}:")
for class_idx, prob in enumerate(probabilities):
    print(f"Class {class_idx}: {prob:.4f}")
```

```
Class probabilities for data point 3:
Class 0: 0.0000
Class 1: 0.0000
Class 2: 0.0000
Class 3: 0.0000
Class 4: 0.0000
Class 5: 0.0000
Class 6: 0.0000
Class 7: 0.0000
Class 8: 1.0000
Class 9: 0.0000
```

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

Yes because they sum up to 1.0

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

[46]:
```
learner_mnist.predict_proba(X[idx:idx+1,:]).round(2)
```

[46]:
```
array([[0., 0., 0., 0., 0., 0., 0., 0., 1., 0.]])
```

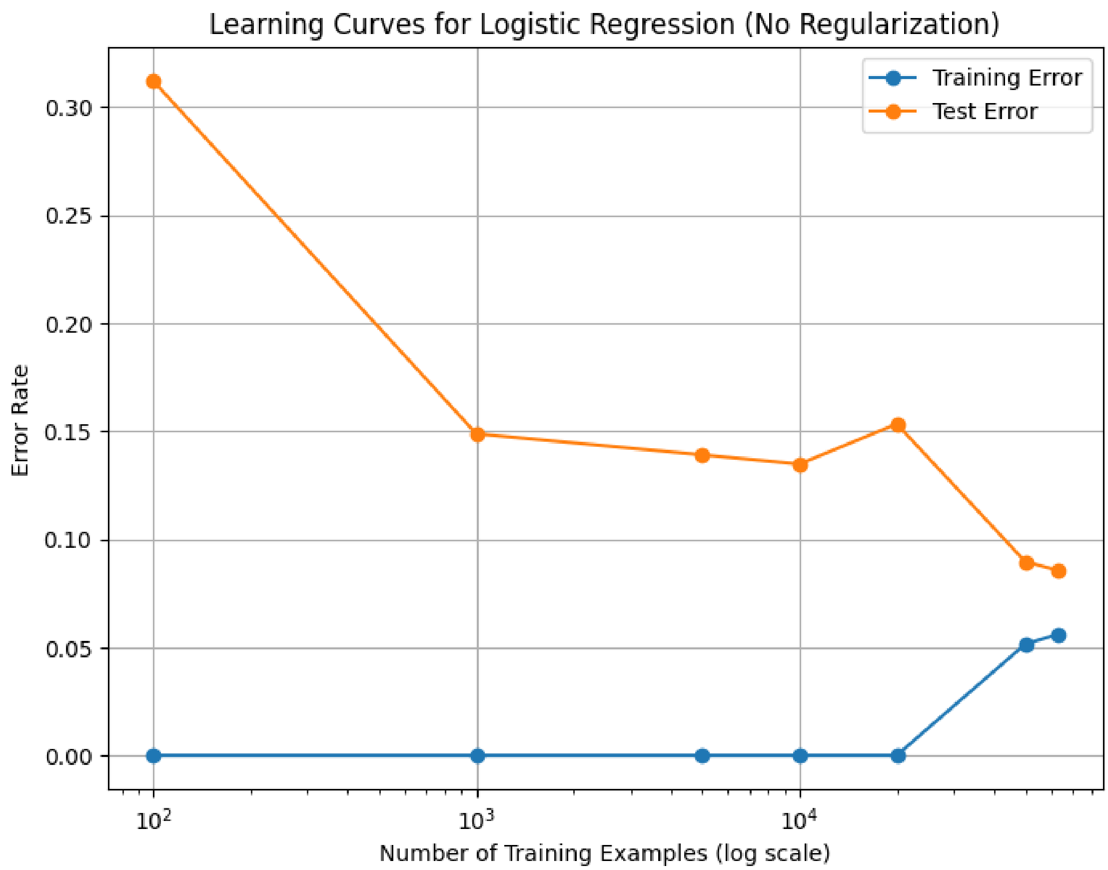
1.7.6 Problem 4.5: Learning Curves (10 points)

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?

[47]:
```
train_sizes = [100, 1000, 5000, 10000, 20000, 50000, 63000]

C = np.inf      # No regularization!

### YOUR CODE STARTS HERE ###
training_errors = []
test_errors = []
# Train a logistic regression model with each data size m and C=infinity
for m_tr in train_sizes:
    # Use the first m_tr samples from the training data
    X_tr_subset = X_tr[:m_tr, :]
    y_tr_subset = y_tr[:m_tr]

    # Train logistic regression with no regularization (C = infinity)
    learner_mnist = LogisticRegression(C=C, random_state=seed, max_iter=1000)
    learner_mnist.fit(X_tr_subset, y_tr_subset)
    # Compute the training and test error rates
    training_errors.append(1 - learner_mnist.score(X_tr_subset, y_tr_subset))
    test_errors.append(1 - learner_mnist.score(X_te, y_te))

plt.figure(figsize=(8, 6))
plt.semilogx(train_sizes, training_errors, label='Training Error', marker='o')
plt.semilogx(train_sizes, test_errors, label='Test Error', marker='o')
plt.xlabel('Number of Training Examples (log scale)')
plt.ylabel('Error Rate')
plt.title('Learning Curves for Logistic Regression (No Regularization)')
plt.legend()
plt.grid(True)
plt.show()
### YOUR CODE ENDS HERE ###
```




```
[ ]: # COMMENT / DISCUSS
```

The plot should show that as the number of training examples increases, both the training and test errors decrease suggesting better generalization.

Although we ran out of data at 63,000 examples, the test error may still show signs of improvement. This suggests that using additional data beyond 63,000 could continue to help improve the model's performance by reducing overfitting

<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 took help from numpy and scikit documentation and some other online tools. Didn't need to discuss with other peers