cs273_hw2

October 10, 2024

1 CS273A Homework 2

1.1 Due Wednesday, October 16th, 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.

1.2.1 Summary of Assignment: 100 total points

- Problem 1: k-Nearest Neighbors (20 points)
  - Problem 1.1: Splitting data into training & test sets (8 points)
  - Problem 1.2: Plot predictions for different values of k (8 points)
  - Problem 1.3: Display performance as a function of k & select best (4 points)
- Problem 2: Linear Regression (20 points)
  - Problem 2.1: Train the model and plot the data along with its predictions (10 points)
  - Problem 2.2: Compute the MSE loss for the training and evaluation data (10 points)

- Problem 3: Feature transformations (20 points)
  - Problem 3.1: Train & display polynomial regression models using feature transforms (10 points)
  - Problem 3.2: Plot the training & evaluation error as a function of degree (5 points)
  - Problem 3.3: Select the best degree for these data (5 points)
- Problem 4: Cross-Validation (20 points)
  - Problem 4.1: Plot the five-fold cross validation error (10 points)
  - Problem 4.2: Select the best degree using cross-validation (5 points)
  - Problem 4.3: Compare cross-validation model selection to hold-out data (5 points)
- Problem 5: Regularization (15 points)
  - Problem 5.1: Train L2-regularized linear regression ('Ridge regression') (5 points)
  - Problem 5.2: Plot MSE as a function of the regularization amount (5 points)
  - Problem 5.3: Select the best amount of regularization (5 points)
- Statement of Collaboration (5 points)

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

from sklearn.model_selection import train_test_split
from sklearn.metrics import zero_one_loss
from sklearn.metrics import mean_squared_error as mse

from sklearn.neighbors import KNeighborsClassifier, KNeighborsRegressor
from sklearn.inspection import DecisionBoundaryDisplay

from sklearn.linear_model import LinearRegression    # Basic Linear Regression
from sklearn.linear_model import Ridge               # Linear Regression with␣
 ↪L2 regularization

from sklearn.model_selection import KFold            # Cross-validation tools

from sklearn.preprocessing import PolynomialFeatures # Feature transformations
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline                # Useful for sequences of␣
 ↪transforms

import requests                                      # reading data
from io import StringIO

seed = 1234
```

2 Training / Test Splits

As we've seen in lecture, it is difficult to tell how accurate our model is from only the data on which it has been trained. For this reason, we usually reserve some data for evaluation, often called "validation" or "test" data. We'll start by loading a one-dimensional regression data set to use in the rest of the homework. We will divide this data set into 75% training data, and 25% evaluation data:

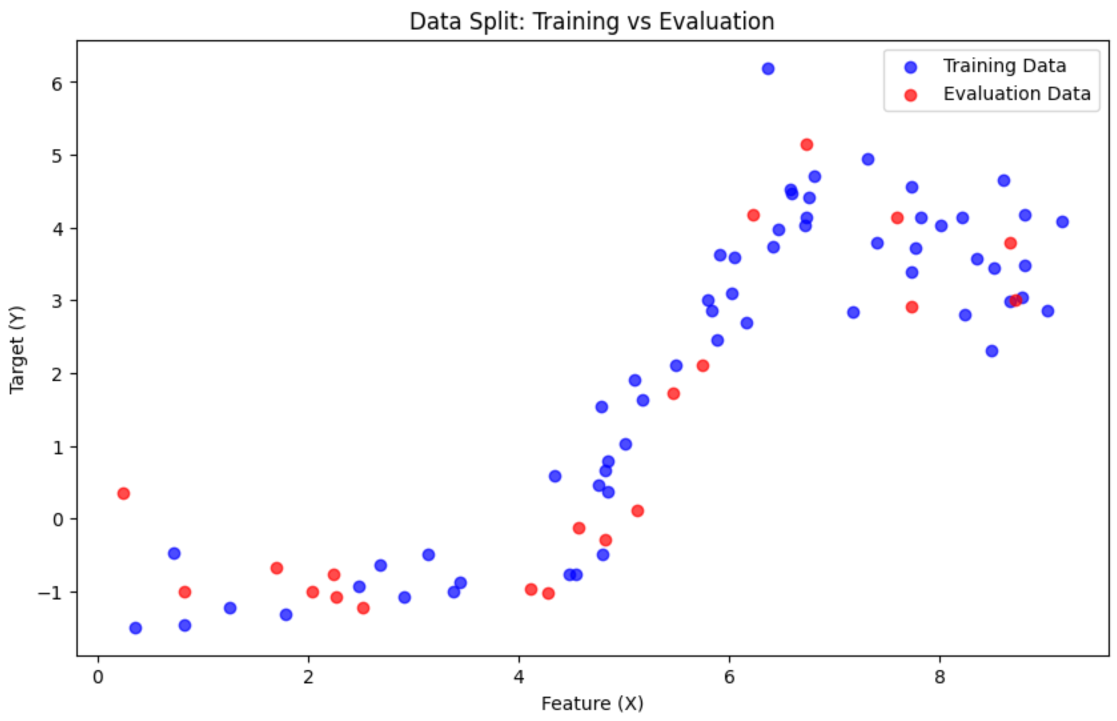
[2]:
```python
url = 'https://www.ics.uci.edu/~ihler/classes/cs273/data/curve80.txt'

with requests.get(url) as link: curve = np.genfromtxt(StringIO(link.
 ↪text),delimiter=None)

X = curve[:,0:-1]       # extract features
Y = curve[:,-1]         # extract target values

# split into training and evaluation data
Xt, Xe, Yt, Ye = train_test_split(X, Y, test_size=0.25, random_state=seed)
```

3 P1: K-Nearest Neighbors Regression

3.0.1 P1.1: Visualizing the Data Splits

Plot the data for this regression problem, with the (scalar) feature x along the horizontal axis, and the real-valued target y as the vertical axis. Plot all the data, displaying the training data X_t in one color, and the evaluation data X_e in a different color.

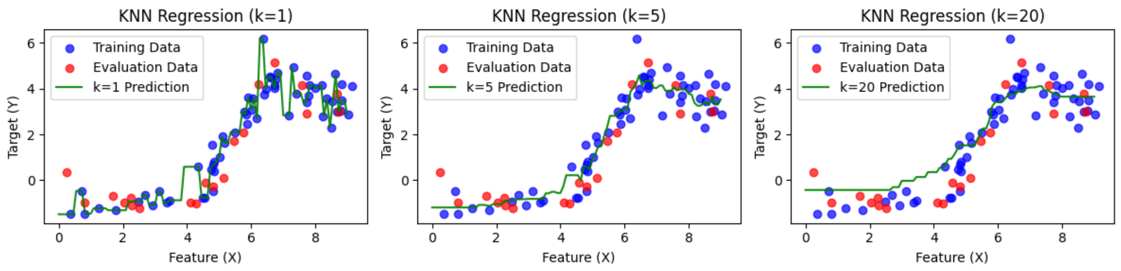
[3]:
```python
plt.figure(figsize=(10, 6))
plt.scatter(Xt, Yt, color='blue', label='Training Data', alpha=0.7)
plt.scatter(Xe, Ye, color='red', label='Evaluation Data', alpha=0.7)
plt.title('Data Split: Training vs Evaluation')
plt.xlabel('Feature (X)')
plt.ylabel('Target (Y)')
plt.legend()
plt.show()
```



3.0.2 P1.2 Visualizing KNN Regression Predictions

Now use sklearn's KNeighborsRegressor class to build a nearest neighbor regression model on your training data. Build three models, using $k = 1$, $k = 5$, and $k = 20$, and for each one display the training data, test data, and prediction function. (Note: you can evaluate the prediction function of your learner by predicting at a dense collection of locations x_spaced along the x-axis, and then predicting at these points and connecting them using plot.)

```
[4]: # Create a figure with 1 row and 3 columns
fig, axes = plt.subplots(1, 3, figsize=(12, 3))

x_spaced = np.linspace(0,9,100).reshape(-1,1)  # get a collection of␣
 ↪x-locations at which to plot f(x)

### YOUR CODE STARTS HERE ###

for i, k in enumerate([1, 5, 20]):
    knn = KNeighborsRegressor(n_neighbors=k)
    knn.fit(Xt, Yt)

    y_spaced_pred = knn.predict(x_spaced)

    axes[i].scatter(Xt, Yt, color='blue', label='Training Data', alpha=0.7)
    axes[i].scatter(Xe, Ye, color='red', label='Evaluation Data', alpha=0.7)
    axes[i].plot(x_spaced, y_spaced_pred, color='green', label=f'k={k}␣
↪Prediction', alpha=0.9)

    axes[i].set_title(f'KNN Regression (k={k})')
    axes[i].set_xlabel('Feature (X)')
    axes[i].set_ylabel('Target (Y)')
    axes[i].legend()



###  YOUR CODE ENDS HERE  ###

fig.tight_layout()
```

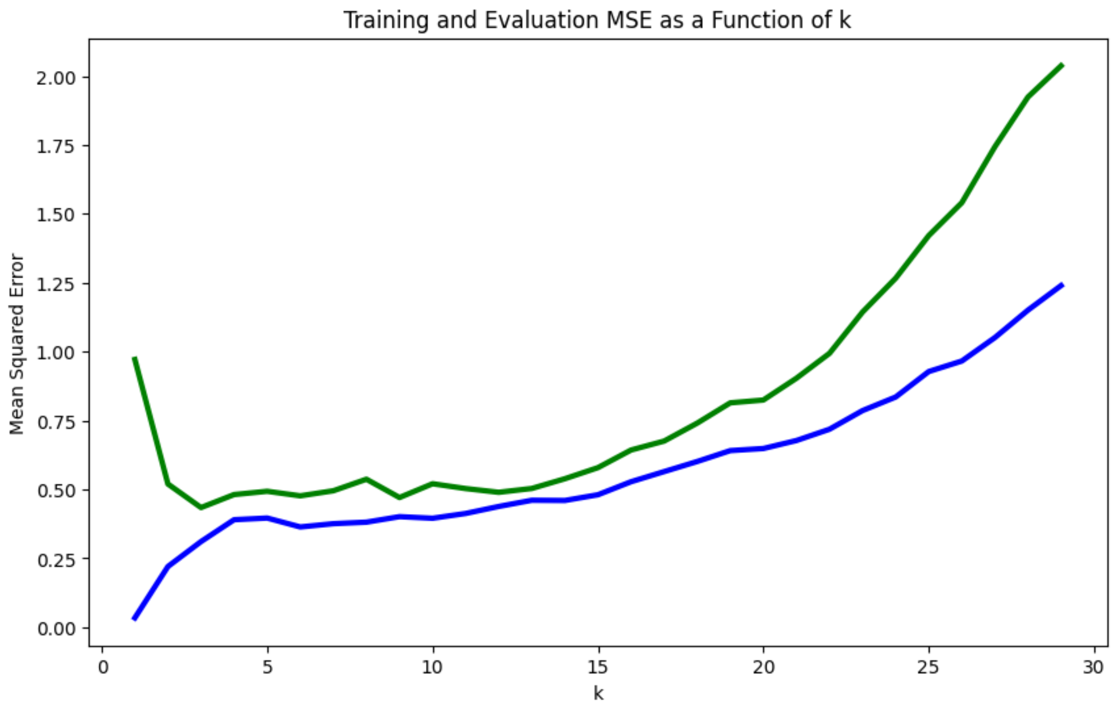


3.0.3 P1.3: KNN Model Selection

Train a model for each k in $1 \leq k \leq 30$, and compute their training and validation MSE. Plot these values as a function of k. What is the best value of k for your model?

[5]:
```
k_values = list(range(1,30))
mse_train = []
mse_eval = []

for i,k in enumerate(k_values):

    ### YOUR CODE STARTS HERE ###
    knn = KNeighborsRegressor(n_neighbors=k)
    knn.fit(Xt, Yt)

    y_train_pred = knn.predict(Xt)
    y_eval_pred = knn.predict(Xe)

    mse_train.append(mse(Yt, y_train_pred))
    mse_eval.append(mse(Ye, y_eval_pred))
    ###  YOUR CODE ENDS HERE  ###
plt.figure(figsize=(10, 6))
plt.xlabel('k')
plt.ylabel('Mean Squared Error')
plt.title('Training and Evaluation MSE as a Function of k')
plt.plot(k_values,mse_train,'b-', k_values,mse_eval,'g-', lw=3)
```

[5]:
```
[<matplotlib.lines.Line2D at 0x7d9bce8c36d0>,
 <matplotlib.lines.Line2D at 0x7d9bce8c37c0>]
```



[6]:
```
best_k = k_values[np.argmin(mse_eval)]
print(f"The best value of k is {best_k}")
```

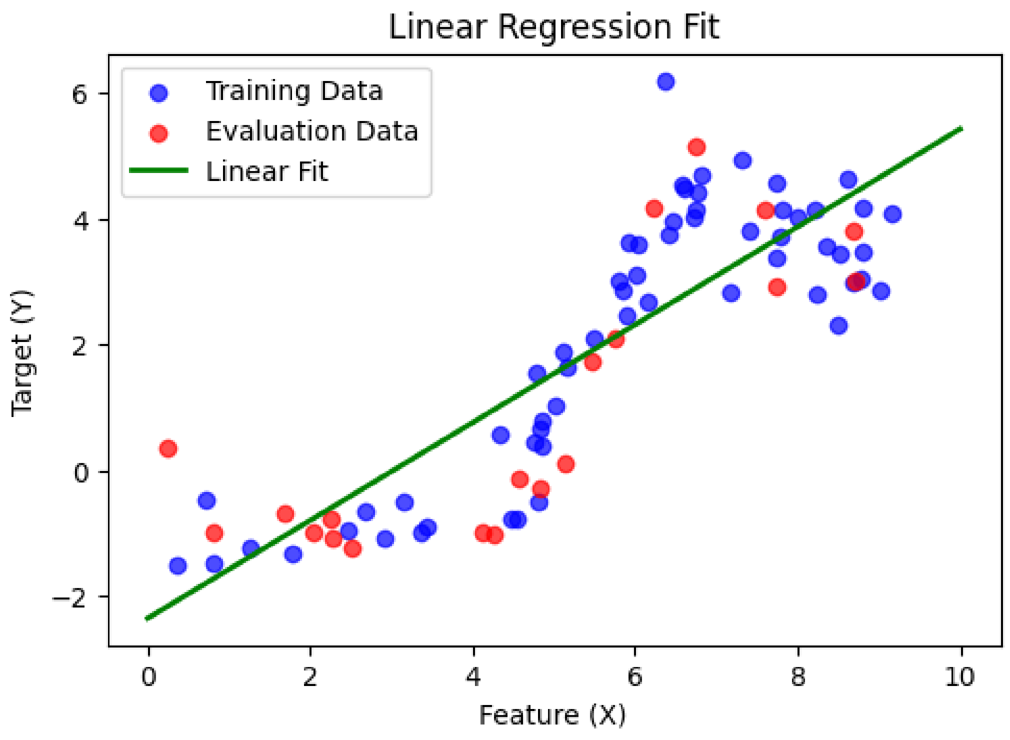
```
The best value of k is 3
```

4 P2: Linear Regression

4.0.1 P2.1: Train linear regression model

Now, let's train a simple linear regression model on the training data. After training the model, plot the training data (colored blue), evaluation data (colored red), and our linear fit (a line) together on a single plot. Also print out the coefficients (slope, `lr.coef_`, and intercept, `lr.intercept_`) of your model after fitting.

[7]:
```
plt.figure(figsize=(6,4))

lr = LinearRegression()
lr.fit(Xt, Yt)

x_spaced = np.linspace(0, 10, 200).reshape(-1, 1)
yhat_spaced = lr.predict(x_spaced)

plt.scatter(Xt, Yt, color='blue', label='Training Data', alpha=0.7)
plt.scatter(Xe, Ye, color='red', label='Evaluation Data', alpha=0.7)
plt.plot(x_spaced, yhat_spaced, color='green', label='Linear Fit', lw=2)

plt.title('Linear Regression Fit')
plt.xlabel('Feature (X)')
plt.ylabel('Target (Y)')
plt.legend()

plt.show()

print(f"Slope (Coefficient): {lr.coef_[0]}")
print(f"Intercept: {lr.intercept_}")
```



```
Slope (Coefficient): 0.7768472052927541
```

```
Intercept: -2.3463013180118275
```

4.0.2 P1.2 Evaluate your model's fit

Compute the mean squared error of your trained model on the training data (the data it was fit on) and the held-out evaluation data.

[8]:
```python
y_train_pred = lr.predict(Xt)
y_eval_pred = lr.predict(Xe)

mse_train = mse(Yt, y_train_pred)
mse_eval = mse(Ye, y_eval_pred)

print(f"Training MSE: {mse_train}")
print(f"Evaluation MSE: {mse_eval}")
```

```
Training MSE: 1.270893125474928
Evaluation MSE: 1.6723519225582435
```

4.1 Problem 3: Feature Transformations

Often we will want to transform our data (as we saw in class). A very simple version of this transformation is "normalizing" the data, in which we shift and scale the feature values to a desirable range; typically, zero mean and unit variance, for example. The `StandardScaler()` object in scikit-learn implements such a transformation.

Typically, a pre-processing transformation works in a similar way to training a model: we `fit` the object to our training data (in this case, computing the empirical mean and variance of the data), and save the parameters of the transformation (the shift and scale values) so that we can apply exactly the same transformation to subsequent data, for example when asked to predict on a new value of x.

So, for example:

[9]:
```python
scale = StandardScaler().fit(Xt)      # find the desired transformation
X_transformed = scale.transform(Xt)   # & apply it to the training data

# Now, we can train our model on X_transformed...
# lr = LinearRegression()...

# Before we predict, we also need to transform the test point's values:
yhat_spaced = lr.predict(scale.transform(x_spaced))
```

If you like (and as described in the Discussion code), you can use `sklearn`'s `Pipeline` object to simplify the process of sequentially applying transformations before a predictor.

[10]:
```python
pipe = Pipeline( [('scale',StandardScaler()),('linreg',LinearRegression())])
pipe.fit(Xt, Yt)
yhat_spaced = pipe.predict(x_spaced)# call transform on each element but last,␣
 ↪then predict on the last
```

4.2 P3.1: Train polynomial regression models

As mentioned in the homework, you can create additional features manually, e.g.,

[11]:
```
m,n = Xt.shape            # rest of this cell assumes n=1 feature
Xt2 = np.zeros((m,2))
Xt2[:,0] = Xt[:,0]
Xt2[:,1] = Xt[:,0]**2
print (Xt.shape)
print (Xt2.shape)
print (Xt2[0:6,:])   # look at a few data points to check:
```

```
(60, 1)
(60, 2)
[[ 0.72580645  0.526795  ]
 [ 2.4769585   6.13532341]
 [ 7.7304147  59.75931143]
 [ 9.0207373  81.37370144]
 [ 8.6751152  75.25762373]
 [ 6.4631336  41.77209593]]
```

or, you can create them using SciKit's PolynomialFeatures transform object:

[12]:
```
Phi = PolynomialFeatures(degree=2,include_bias=False).fit(Xt)
Xt2 = Phi.transform(Xt)
print (Xt2[0:6,:])   # look at the same data points -- same values
```

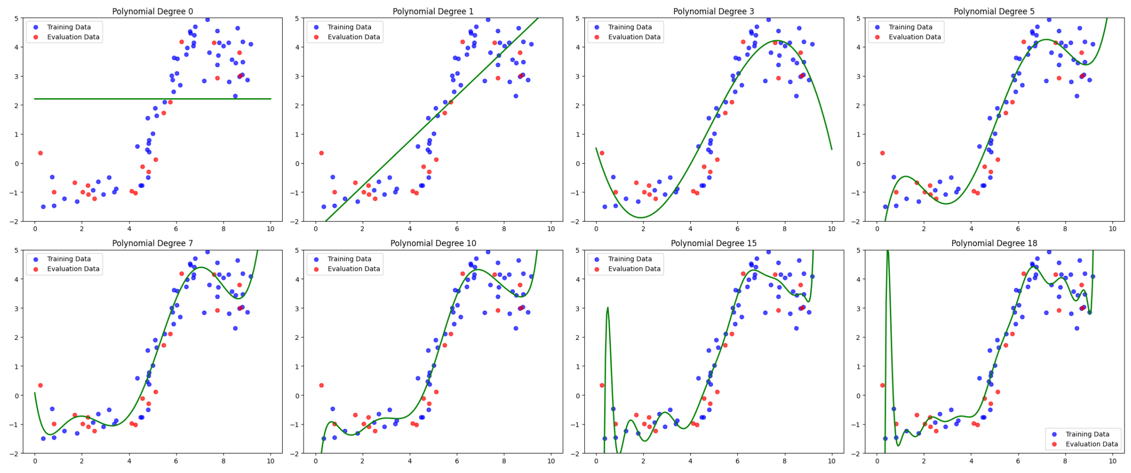
```
[[ 0.72580645  0.526795  ]
 [ 2.4769585   6.13532341]
 [ 7.7304147  59.75931143]
 [ 9.0207373  81.37370144]
 [ 8.6751152  75.25762373]
 [ 6.4631336  41.77209593]]
```

Now, try fitting a linear regression model using different numbers of polynomial features of x.

For each degree $d \in \{0, 1, 3, 5, 7, 10, 15, 18\}$:

- Fit a linear regression model using features consisting of all powers of x up to degree d
  - Make sure you apply `StandardScaler` to the transformed data before training
- Plot the resulting prediction function $f(x)$, along with the training and validation data as before

[13]:
```
degrees = [0,1,3,5,7,10,15,18]
learners = [ [] ]*len(degrees)

fig, ax = plt.subplots(2,4, figsize=(24,10))

for i,degree in enumerate(degrees):

    ### YOUR CODE STARTS HERE ###
    Phi = PolynomialFeatures(degree=degree, include_bias=True)
    Xt_poly = Phi.fit_transform(Xt)
    Xe_poly = Phi.transform(Xe)

    scaler = StandardScaler().fit(Xt_poly)
    Xt_scaled = scaler.transform(Xt_poly)
    Xe_scaled = scaler.transform(Xe_poly)

    lr = LinearRegression()
    lr.fit(Xt_scaled, Yt)
    learners[i] = lr

    x_spaced_poly = Phi.transform(x_spaced)
    yhat_spaced = lr.predict(scaler.transform(x_spaced_poly))

    axi = ax[i // 4, i % 4]
    axi.scatter(Xt, Yt, color='blue', label='Training Data', alpha=0.7)
    axi.scatter(Xe, Ye, color='red', label='Evaluation Data', alpha=0.7)
    axi.plot(x_spaced, yhat_spaced, color='green', lw=2)

    axi.set_ylim([-2, 5])  # consistent y-scale for comparison
    axi.set_title(f'Polynomial Degree {degree}')
    axi.legend()

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

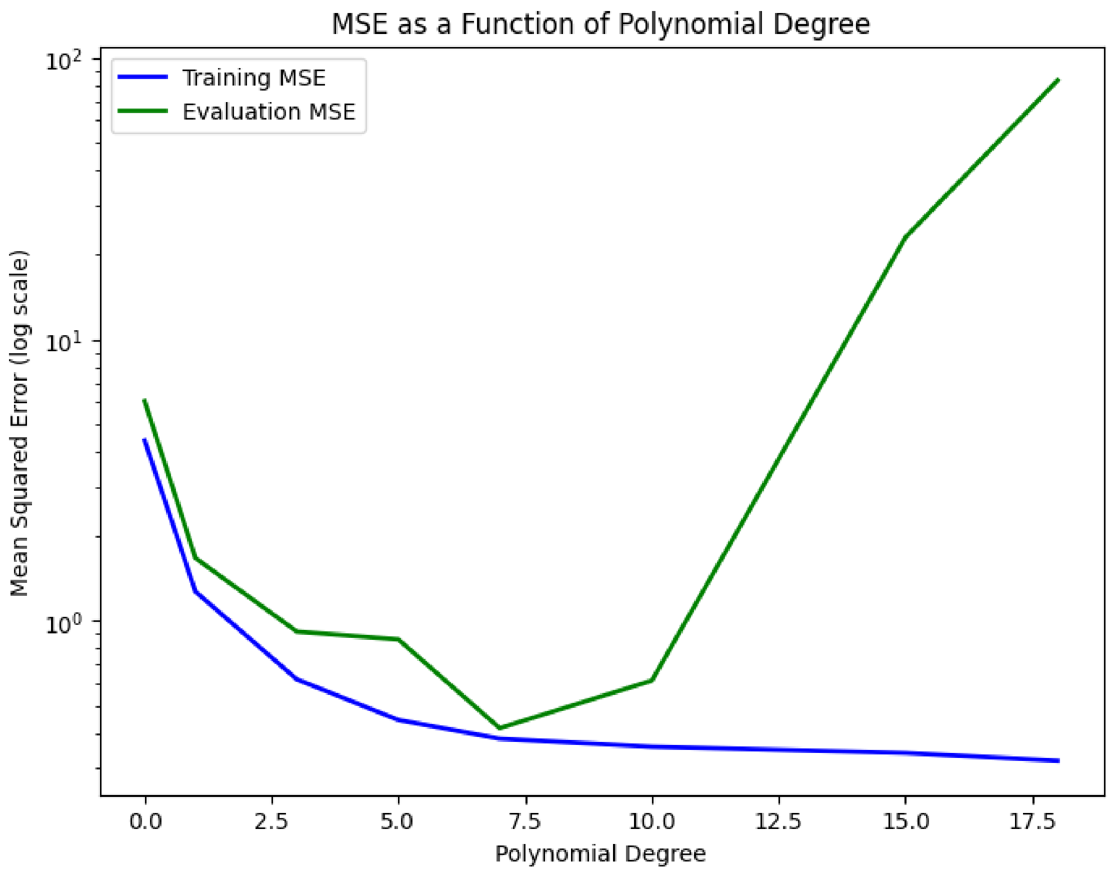


4.3 P3.2 Model Performance

Compute the mean squared error (MSE) loss of each of your trained models on both the training data and the evaluation data. Plot these errors as a function of degree (so, degree along the horizontal axis, MSE loss as the vertical axis).

[14]:
```
mse_train = [0]*len(degrees)
mse_test = [0]*len(degrees)

for i,degree in enumerate(degrees):
    # Recompute the degree-d polynomial transform
    Phi = PolynomialFeatures(degree=degree, include_bias=True)

    Xt_poly = Phi.fit_transform(Xt)
    Xe_poly = Phi.transform(Xe)

    # Rescale the transformed data
    scaler = StandardScaler().fit(Xt_poly)
    Xt_scaled = scaler.transform(Xt_poly)
    Xe_scaled = scaler.transform(Xe_poly)

    # Predict using the trained model
    y_train_pred = learners[i].predict(Xt_scaled)
    y_test_pred = learners[i].predict(Xe_scaled)

    # Compute MSE for training and evaluation data
    mse_train[i] = mse(Yt, y_train_pred)
    mse_test[i] = mse(Ye, y_test_pred)

# Plot the results
plt.figure(figsize=(8, 6))
plt.semilogy(degrees, mse_train, 'b-', label='Training MSE', lw=2)
plt.semilogy(degrees, mse_test, 'g-', label='Evaluation MSE', lw=2)
plt.xlabel('Polynomial Degree')
plt.ylabel('Mean Squared Error (log scale)')
plt.title('MSE as a Function of Polynomial Degree')
plt.legend()
```

[14]:
```
<matplotlib.legend.Legend at 0x7d9bccfb51b0>
```



4.4 P3.3 Model Selection

Which degree would you select to use?

```
[15]: best_degree = degrees[np.argmin(mse_test)]
      print(f"The best degree to use is {best_degree}")
```

```
The best degree to use is 7
```

5 P4: Cross-validation

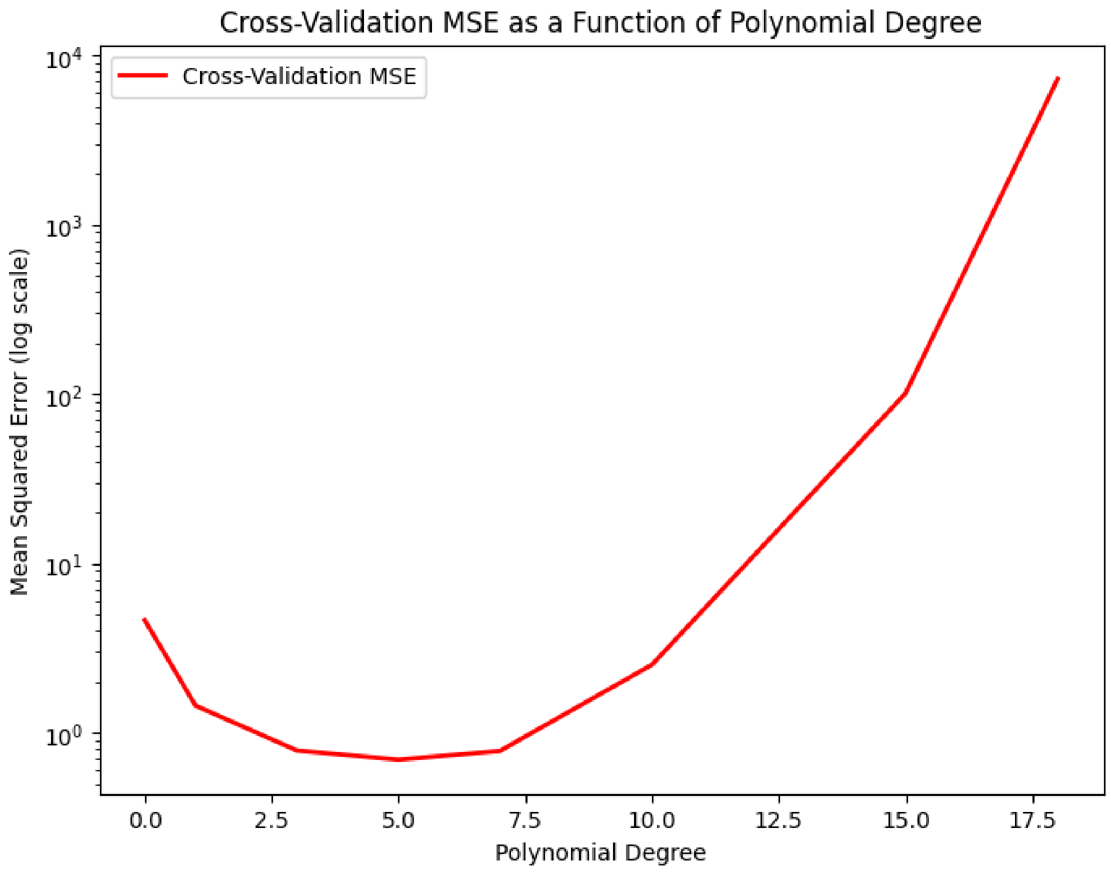
Cross validation is another method of model complexity assessment. We use it only to determine the correct setting of complexity-altering parameters ("hyperparameters"), such as how many and which features to use, or parameters like "k" in KNN, for which training error alone provides little information. In particular, cross validation will not produce a specific model (parameter values), only a setting of the hyperparameter values that cross-validation thinks will lead to a model (parameter values) with low test error.

5.1 P4.1: 5-Fold Cross-validation

In the previous problem, we decided what degree of polynomial fit to use based on the performance on a held-out set of test data. Now suppose that we do not have access to the target values of those data. How can we determine the best degree?

We could perform another split; but since this is reducing the number of data available, let us instead use cross-validation to evaluate the degrees. Cross-validation works by splitting the training data X_T multiple times, one for eack of the K partitions (`n_splits` in the code), and repeat our entire training and evaluation procedure on each split:

[17]:
```
mse_xval = [ 0. ]*len(degrees)

for j,degree in enumerate(degrees):   # loop over desired degree values

    ### YOUR CODE STARTS HERE ###

    xval = KFold(n_splits = 5)      # split into k=5 splits

    mse_fold = []

    for train_index, val_index in xval.split(Xt):
        Xti, Xvi = Xt[train_index], Xt[val_index]
        Yti, Yvi = Yt[train_index], Yt[val_index]

        # Create polynomial feature expansion
        Phi = PolynomialFeatures(degree=degree, include_bias=True)
        Xti_poly = Phi.fit_transform(Xti)
        Xvi_poly = Phi.transform(Xvi)

        # Create StandardScaler
        scaler = StandardScaler().fit(Xti_poly)
        Xti_scaled = scaler.transform(Xti_poly)
        Xvi_scaled = scaler.transform(Xvi_poly)

        # Train the linear regression model
        lr = LinearRegression()
        lr.fit(Xti_scaled, Yti)

        # Predict and compute MSE on validation fold
        Yvi_pred = lr.predict(Xvi_scaled)
        mse_fold.append(mse(Yvi, Yvi_pred))

    # Average MSE across the five folds for this degree
    mse_xval[j] = np.mean(mse_fold)

# Plot the estimated MSE from cross-validation as a function of the degree
plt.figure(figsize=(8, 6))
plt.semilogy(degrees, mse_xval, 'r-', label='Cross-Validation MSE', lw=2)
plt.xlabel('Polynomial Degree')
plt.ylabel('Mean Squared Error (log scale)')
plt.title('Cross-Validation MSE as a Function of Polynomial Degree')
plt.legend()
plt.show()

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



5.2 P4.2: Cross-validation model selection

What degree would you choose based on the cross validation performance?

[18]:
```
best_degree_cv = degrees[np.argmin(mse_xval)]
print(f"The best degree based on cross-validation is {best_degree_cv}")
```

```
The best degree based on cross-validation is 5
```

5.3 P4.3 Comparison to test performance

How do the MSE estimates from 5-fold cross-validation compare to the estimated test performance you found from your held-out data, X_E? Explain briefly.

[19]:
```python
Phi = PolynomialFeatures(degree=best_degree_cv, include_bias=True)
Xt_poly = Phi.fit_transform(Xt)
Xe_poly = Phi.transform(Xe)


scaler = StandardScaler().fit(Xt_poly)
Xt_scaled = scaler.transform(Xt_poly)
Xe_scaled = scaler.transform(Xe_poly)

lr = LinearRegression()
lr.fit(Xt_scaled, Yt)

mse_train_final = mse(Yt, lr.predict(Xt_scaled))

mse_test_final = mse(Ye, lr.predict(Xe_scaled))

print(f"Cross-validation MSE (for best degree): {min(mse_xval)}")
print(f"Test MSE (held-out data XE): {mse_test_final}")
```

```
Cross-validation MSE (for best degree): 0.6942723737403853
Test MSE (held-out data XE): 0.8597344241311925
```

Best Degree from Cross-validation: We identify the degree that minimizes the cross-validation MSE.

Train Model: The model is retrained using the best degree on the entire training set (Xt).

MSE Comparison: The MSE from cross-validation (min(mse_xval)) is printed for the best degree. The MSE on the held-out test set (XE) is computed and printed.

6 P5 : Regularization

In systems where we already have a lot of features, or where we do not know which of the many features we might construct will be helpful, we can use regularization to help us control overfitting.

6.1 P5.1: Regularized Regression

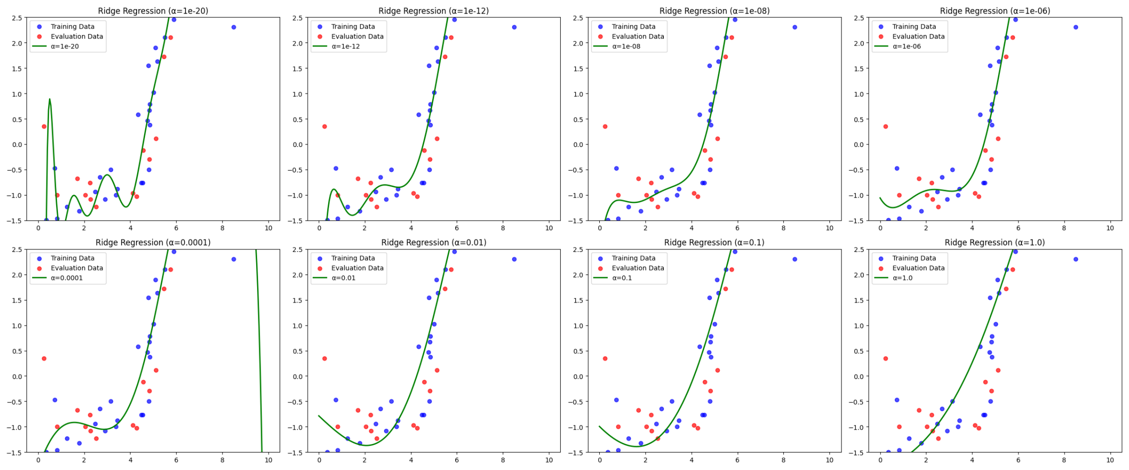
In `sklearn`, linear regression with quadratic (L2) regularization is implemented in a different object, `Ridge`. Use this ridge regression model to fit your degree-18 data using various amounts of regularization:

$$\alpha \in \{10^{-20}, 10^{-12}, 10^{-8}, 10^{-6}, 10^{-4}, 0.01, 0.1, 1.0\}$$

Plot the training and evaluation data, along with the predicted regression function for each value.

[21]:
```python
alphas = [1e-20, 1e-12, 1e-8, 1e-6, 1e-4, 1e-2, 1e-1, 1.]
learners = [ None ]*len(alphas)
```

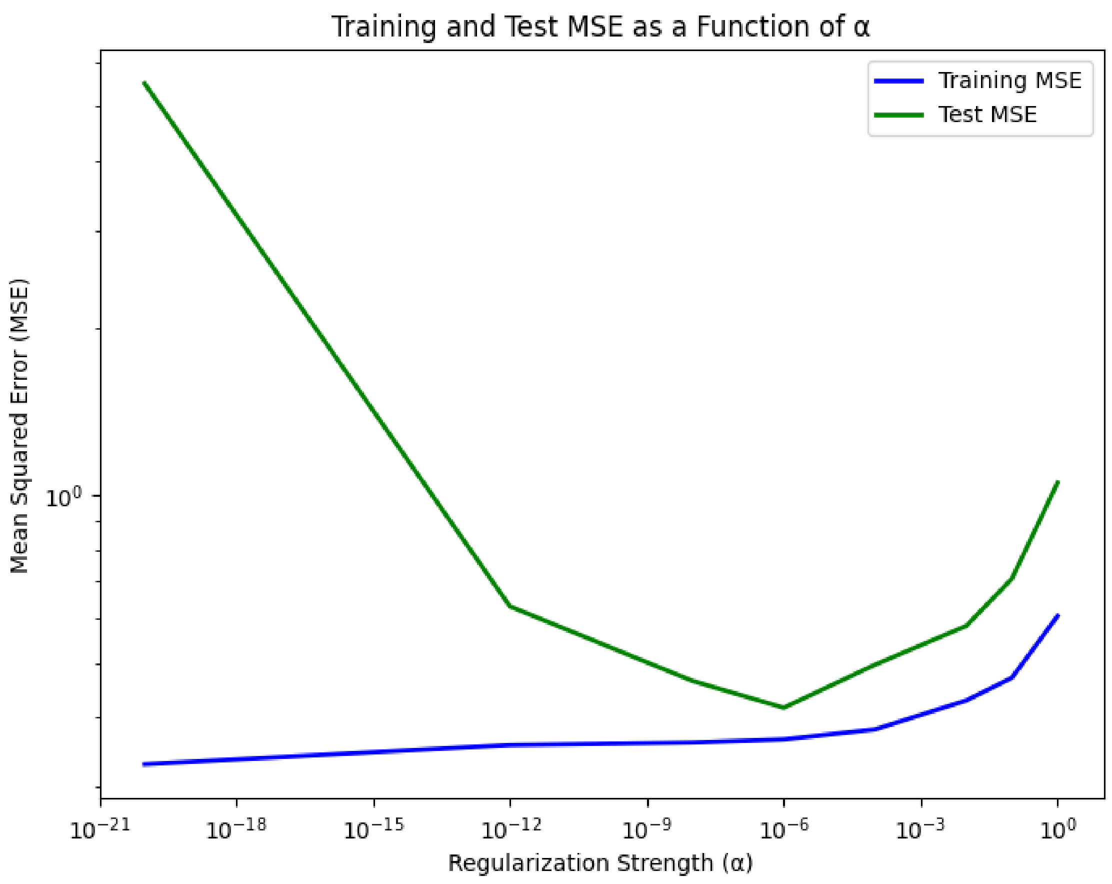
```python
fig, ax = plt.subplots(2,4, figsize=(24,10))
degree = 18
Phi = PolynomialFeatures(degree=degree, include_bias=False)

Xt_poly = Phi.fit_transform(Xt)
Xe_poly = Phi.transform(Xe)

scaler = StandardScaler().fit(Xt_poly)
Xt_scaled = scaler.transform(Xt_poly)
Xe_scaled = scaler.transform(Xe_poly)

x_spaced = np.linspace(0, 10, 200).reshape(-1, 1)
x_spaced_poly = Phi.transform(x_spaced)
x_spaced_scaled = scaler.transform(x_spaced_poly)

for i,alpha in enumerate(alphas):

    ### YOUR CODE STARTS HERE ###
    pipe = Pipeline([
        ('ridge', Ridge(alpha=alpha))
    ])


    pipe.fit(Xt_scaled, Yt)
    learners[i] = pipe


    yhat_spaced = pipe.predict(x_spaced_scaled)


    axi = ax[i // 4, i % 4]
    axi.scatter(Xt, Yt, color='blue', label='Training Data', alpha=0.7)
    axi.scatter(Xe, Ye, color='red', label='Evaluation Data', alpha=0.7)
    axi.plot(x_spaced, yhat_spaced, color='green', label=f' ={alpha}', lw=2)

    axi.set_ylim([-1.5, 2.5])
    axi.set_title(f'Ridge Regression ( ={alpha})')
    axi.legend()

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



6.2 P5.2: Training and Test Performance

Using your trained models, evaluate the training and test MSE as a function of the regularization α. Plot these functions. (It is best to use a log-scale for both alpha and MSE, for clarity.)

[22]:
```
mse_train = [0]*len(alphas)
mse_test = [0]*len(alphas)

for i,alpha in enumerate(alphas):
    y_train_pred = learners[i].predict(Xt_scaled)
    y_test_pred = learners[i].predict(Xe_scaled)

    mse_train[i] = mse(Yt, y_train_pred)
    mse_test[i] = mse(Ye, y_test_pred)

plt.figure(figsize=(8, 6))
plt.loglog(alphas, mse_train, 'b-', label='Training MSE', lw=2)
plt.loglog(alphas, mse_test, 'g-', label='Test MSE', lw=2)
plt.xlabel('Regularization Strength ( )')
plt.ylabel('Mean Squared Error (MSE)')
plt.title('Training and Test MSE as a Function of  ')
plt.legend()
plt.show()
```



6.3 P5.3: Model Selection

Which regularization value α would you select? Identify in which regions α is underfitting or overfitting.

[23]:
```
# Identify the best alpha based on the lowest test MSE
best_alpha_index = np.argmin(mse_test)
best_alpha = alphas[best_alpha_index]

print(f"The best regularization value   is {best_alpha}")
```

```
The best regularization value   is 1e-06
```

Regions of Overfitting and Underfitting:

Overfitting (small):

In the region where the training MSE is significantly lower than the test MSE. Regions where   is less than 1e-15.

Underfitting (large):

In the region where both training and test MSEs are relatively high and close to each other, suggesting the model is too simple. Region where is close to 1 or around.

6.3.1 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 didn't discuss with other students. I took help from the provided lecture notes and some online blogs