CS273A Homework 2

Due Wednesday, October 16th, 11:59pm

Name: Langtian Qin
Student ID: 80107838
Email: langtiq@uci.edu

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: 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)

In [31]:
```
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
```

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:

In [7]:
```
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
```

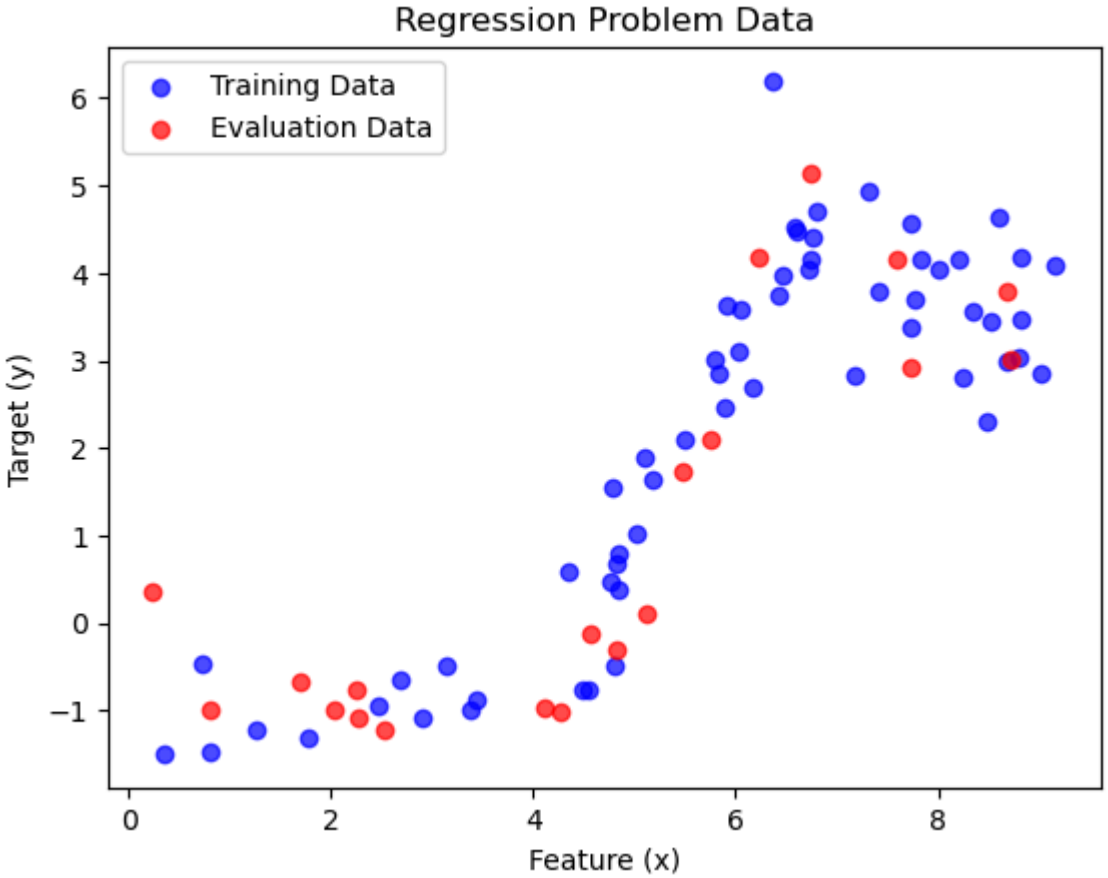
```
# split into training and evaluation data
Xt, Xe, Yt, Ye = train_test_split(X, Y, test_size=0.25, random_state=seed)
```

P1: K-Nearest Neighbors Regression

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.

In [11]:
```
# Plot the data
plt.scatter(Xt, Yt, color='blue', label='Training Data', alpha=0.7)
plt.scatter(Xe, Ye, color='red', label='Evaluation Data', alpha=0.7)

# Set plot labels and title
plt.xlabel('Feature (x)')
plt.ylabel('Target (y)')
plt.title('Regression Problem Data')
plt.legend()

# Show the plot
plt.show()
```

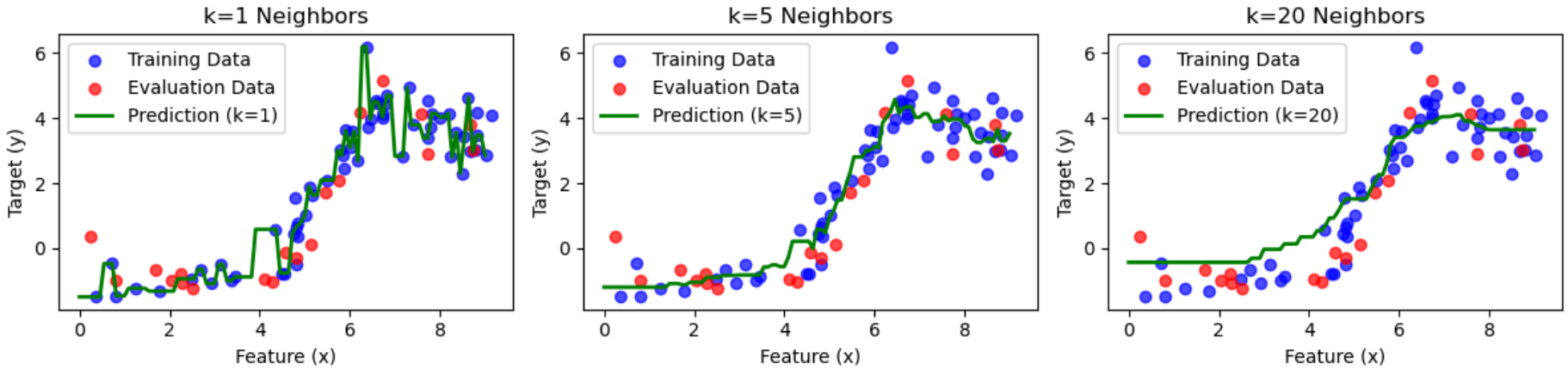


P1.2 Visualizing KNN Regression Predictions

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`.)

In [14]:
```
# 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 k in [1,5,20]:
    knn = KNeighborsRegressor(n_neighbors=k)  # Create KNeighborsRegressor with k neighbors

    knn.fit(Xt, Yt)  # Fit the model on the training data

    y_pred = knn.predict(x_spaced)  # Predict on the dense collection of x-locations

    # Get the current axis for plotting
    ax = axes[[1, 5, 20].index(k)]

    # Plot the training data
    ax.scatter(Xt, Yt, color='blue', label='Training Data', alpha=0.7)

    # Plot the evaluation data
    ax.scatter(Xe, Ye, color='red', label='Evaluation Data', alpha=0.7)

    # Plot the prediction function
    ax.plot(x_spaced, y_pred, color='green', label=f'Prediction (k={k})', linewidth=2)

    # Set title and labels
    ax.set_title(f'k={k} Neighbors')
    ax.set_xlabel('Feature (x)')
    ax.set_ylabel('Target (y)')
    ax.legend()



###  YOUR CODE ENDS HERE  ###

fig.tight_layout()
```



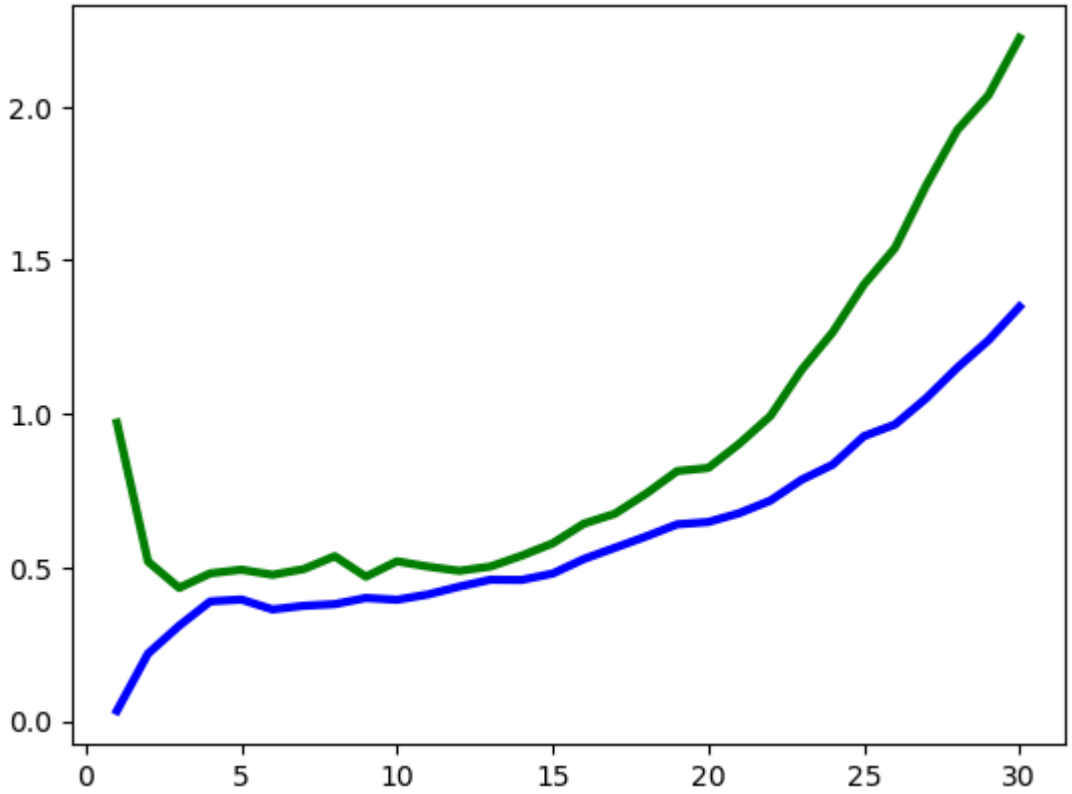
P1.3: KNN Model Selection

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?

In [17]:
```
k_values = list(range(1,31))  # range(1,31) or range(1,30)???
mse_train = []
mse_eval = []

for i,k in enumerate(k_values):

    ### YOUR CODE STARTS HERE ###
    knn = KNeighborsRegressor(n_neighbors=k)
    # Fit the model to the training data
    knn.fit(Xt, Yt)

    # Predict on the training set and compute MSE
    y_train_pred = knn.predict(Xt)
    train_mse = mse(Yt, y_train_pred)
    mse_train.append(train_mse)

    # Predict on the evaluation set and compute MSE
    y_eval_pred = knn.predict(Xe)
    eval_mse = mse(Ye, y_eval_pred)
    mse_eval.append(eval_mse)
```

```
# Find the index of the minimum MSE in the evaluation set
min_eval_mse = min(mse_eval)
best_k = k_values[mse_eval.index(min_eval_mse)]
print("The best K for the evaluation set:", best_k)

    ###  YOUR CODE ENDS HERE  ###
plt.plot(k_values,mse_train,'b-', k_values,mse_eval,'g-', lw=3);
```

```
The best K for the evaluation set: 3
```



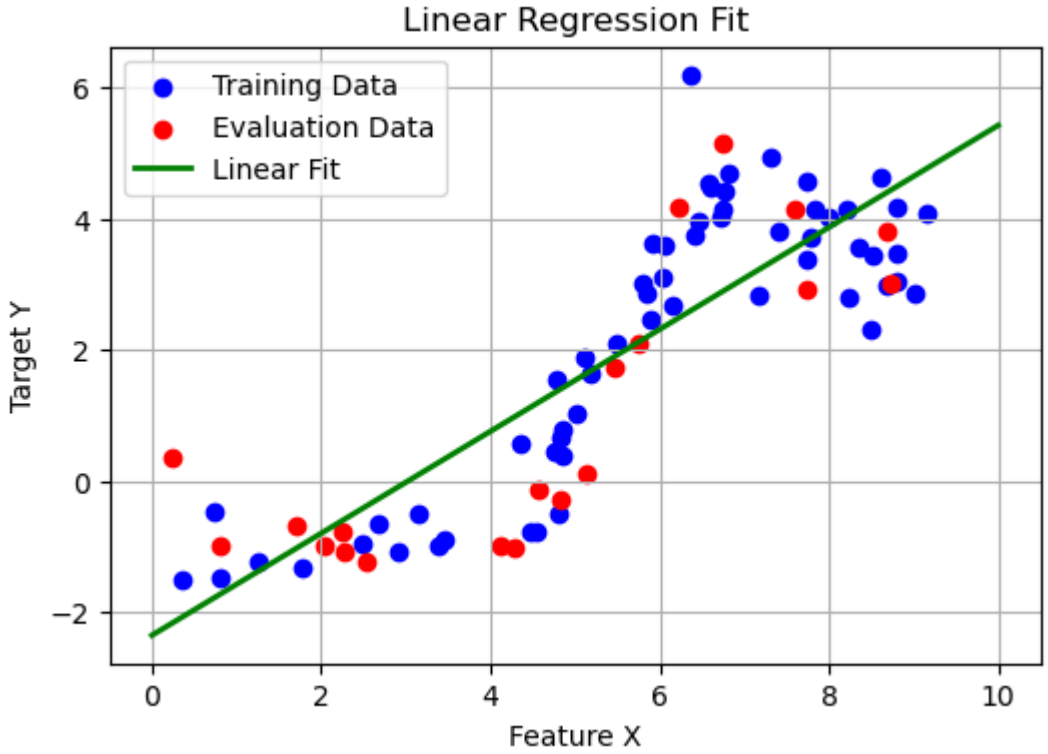
P2: Linear Regression

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.

In [21]:
```
plt.figure(figsize=(6,4))

    ### YOUR CODE STARTS HERE ###

lr = LinearRegression()      # create and fit model to training data
lr.fit(Xt, Yt)           # Fit the model to the training data

# to plot the prediction, we'll evaluate our model at a dense set of locations:
x_spaced = np.linspace(0,10,200).reshape(-1,1)   # data points should be shape (m,1)
yhat_spaced = lr.predict(x_spaced)

# Plot the training data, evaluation data, and linear fit
plt.figure(figsize=(6,4))
plt.scatter(Xt, Yt, color='blue', label='Training Data')   # Plot training data in blue
plt.scatter(Xe, Ye, color='red', label='Evaluation Data')  # Plot evaluation data in red
plt.plot(x_spaced, yhat_spaced, color='green', linewidth=2, label='Linear Fit')  # Linear fit line

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

print("Slope:",lr.coef_)  # Print the slope of the linear fit
print("Intercept:",lr.intercept_)  # Print the intercept of the linear fit

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

```
Slope: [0.77684721]
Intercept: -2.3463013180118275
<Figure size 600x400 with 0 Axes>
```



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.

In [24]:
```
# Predict on the training data
y_train_pred = lr.predict(Xt)

# Calculate MSE on the training data
mse_train = mse(Yt, y_train_pred)

# Predict on the evaluation data
y_eval_pred = lr.predict(Xe)

# Calculate MSE on the evaluation data
mse_eval = mse(Ye, y_eval_pred)
```

```
# Print the MSE results
print("Mean Squared Error on Training Data:", mse_train)
print("Mean Squared Error on Evaluation Data:", mse_eval)
```

```
Mean Squared Error on Training Data: 1.270893125474928
Mean Squared Error on Evaluation Data: 1.6723519225582435
```

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:

In [28]:
```
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.

In [35]:
```
pipe = Pipeline( [('scale',StandardScaler()),('linreg',LinearRegression())])
pipe.fit(Xt, Yt)     # call fit on each element in the pipeline
```

```
pipe.predict(Xe) # call transform on each element but last, then predict on the last
```

Out[35]:
```
array([ 0.97409396,  4.42873713,  1.63638308,  0.84879604, -0.76217741,
        1.20679015,  2.49556894,  2.88936242, -0.60108005,  4.39293768,
        3.55165154,  3.65904974, -0.58318032, -1.03067298,  2.11967511,
       -1.71086178, -0.38628358,  1.90487864,  1.40368689, -2.15835442])
```

P3.1: Train polynomial regression models

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

In [38]:
```
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:

In [41]:
```
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]]
```

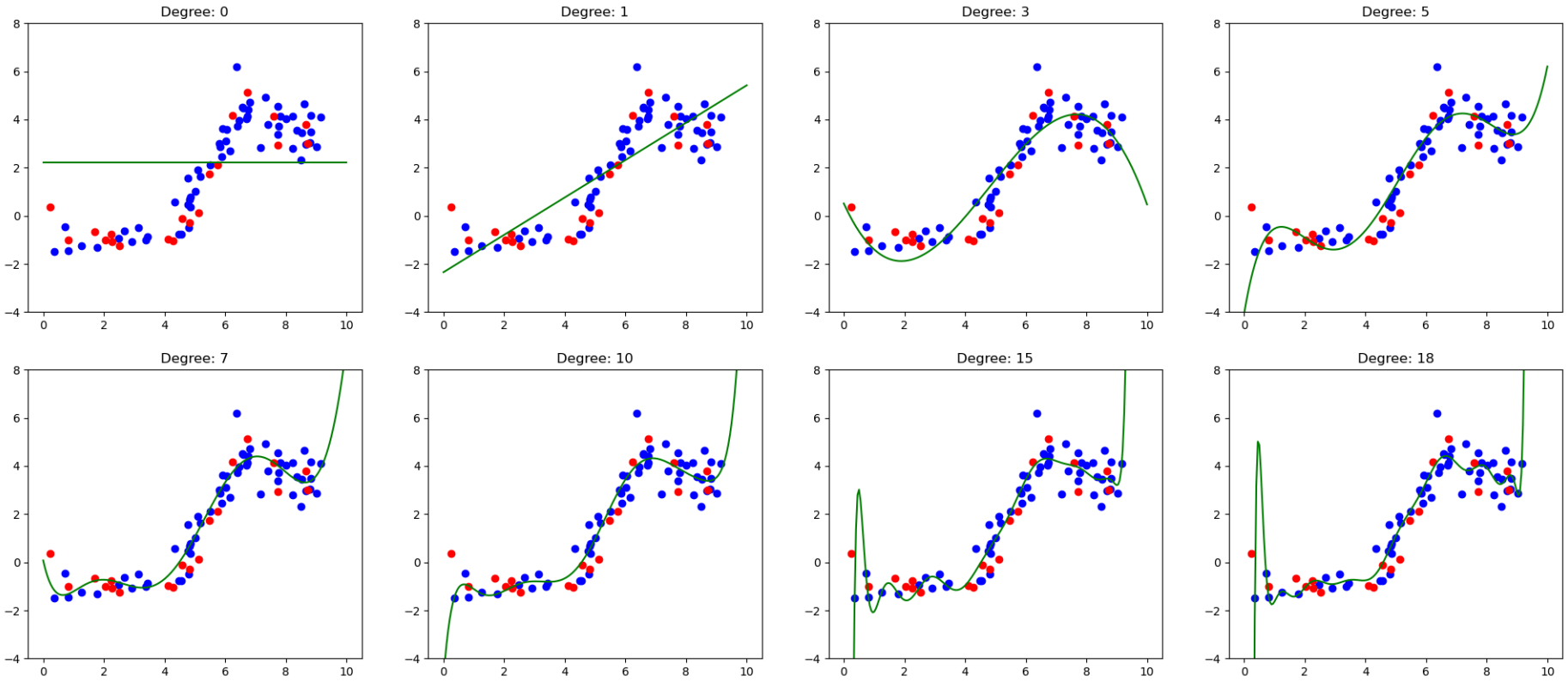
Now, try fitting a linear regression model using different numbers of polynomial features of x.

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

In [44]:
```
degrees = [0,1,3,5,7,10,15,18]
learners = [ [] ]*len(degrees)

fig, ax = plt.subplots(2,4, figsize=(24,10))
x_spaced = np.linspace(0,10,200).reshape(-1,1)   # data points should be shape (m,1)

for i,degree in enumerate(degrees):

    ### YOUR CODE STARTS HERE ###
    if degree == 0:
        m,n = Xt.shape
        Xt_poly = np.zeros((m,1))
        Xt_poly[:,0] = Xt[:,0]**0

        m,n = Xe.shape
        Xe_poly = np.zeros((m,1))
        Xe_poly[:,0] = Xe[:,0]**0

        m,n = x_spaced.shape
        Xs_poly = x_spaced**0

    else:
        # Create a polynomial feature expansion of degree d
        Phi_t = PolynomialFeatures(degree=degree, include_bias=False).fit(Xt)
```

```
        Xt_poly = Phi_t.transform(Xt)

        Phi_e = PolynomialFeatures(degree=degree, include_bias=False).fit(Xe)
        Xe_poly = Phi_e.transform(Xe)

        Phi_s = PolynomialFeatures(degree=degree, include_bias=False).fit(x_spaced)
        Xs_poly = Phi_s.transform(x_spaced)

    # Use StandardScaler to rescale the transformed data
    scale = StandardScaler().fit(Xt_poly)     # find the desired transformation
    Xt_transformed = scale.transform(Xt_poly)  # & apply it to the training data
    Xe_transformed = scale.transform(Xe_poly)  # & apply it to the eval data
    Xs_transformed = scale.transform(Xs_poly)  # & apply it to the eval data

    # Fit your linear regression and save it to "learners"
    lr = LinearRegression()      # create and fit model to training data
    lr.fit(Xt_transformed, Yt)           # Fit the model to the training data
    yt_pred = lr.predict(Xt_transformed)
    ys_pred = lr.predict(Xs_transformed)
    learners[i] = lr

    axi = ax[i//4,i%4]
    # plot the data and your prediction function
    axi.plot(Xt, Yt, 'bo', label='Training Data')  # Blue dots for training data
    axi.plot(Xe, Ye, 'ro', label='Evaluation Data')  # Red dots for evaluation data
    axi.plot(x_spaced, ys_pred, 'g-', label='Prediction')  # Green line for the prediction function
    axi.set_ylim(-4, 8)        # you'll want to set a consistent y-scale for comparison
    axi.set_title(f'Degree: {degree}')  # don't forget to label your plots

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

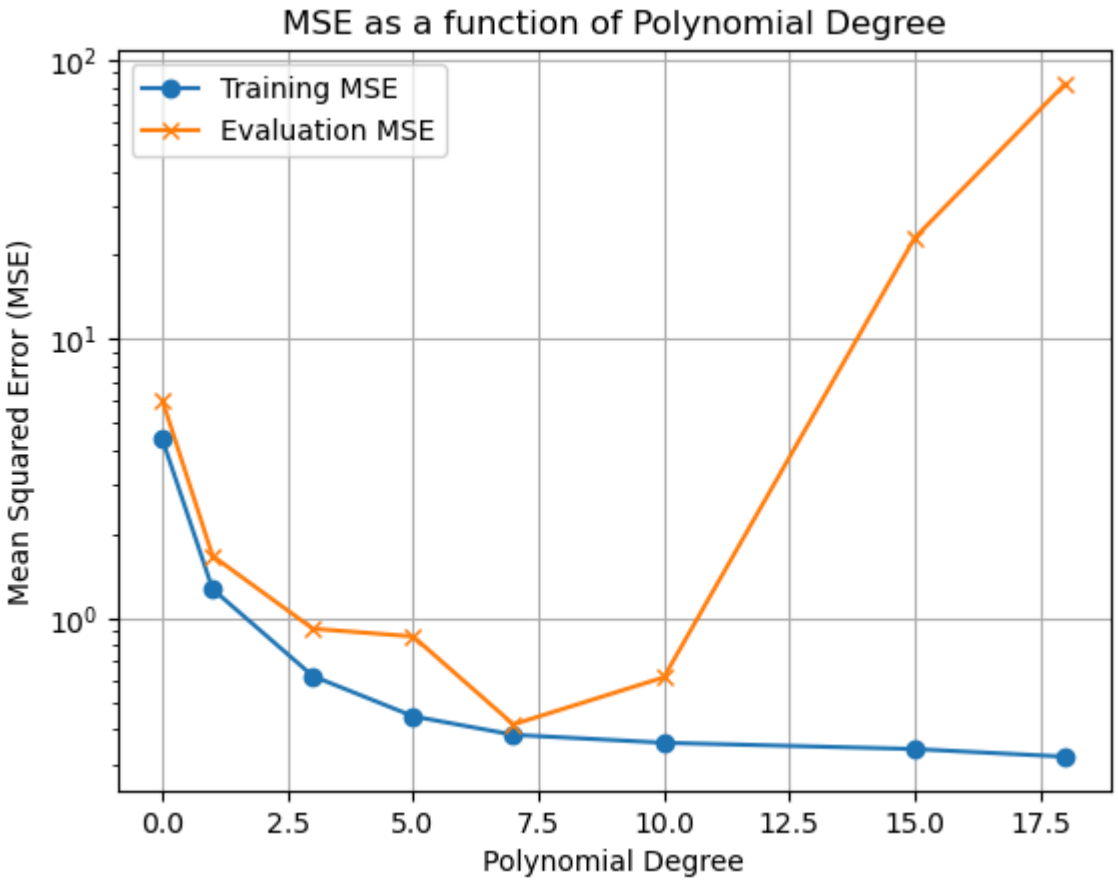


P3.2 Model Performance

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).

In [47]:
```
mse_train = [0]*len(degrees)
mse_test = [0]*len(degrees)

for i,degree in enumerate(degrees):
    if degree == 0:
        m,n = Xt.shape
        Xt_poly = np.zeros((m,1))
        Xt_poly[:,0] = Xt[:,0]**0
```

```
        m,n = Xe.shape
        Xe_poly = np.zeros((m,1))
        Xe_poly[:,0] = Xe[:,0]**0
    else:
        # Create a polynomial feature expansion of degree d
        Phi_t = PolynomialFeatures(degree=degree, include_bias=False).fit(Xt)
        Xt_poly = Phi_t.transform(Xt)

        Phi_e = PolynomialFeatures(degree=degree, include_bias=False).fit(Xe)
        Xe_poly = Phi_e.transform(Xe)

    # Use StandardScaler to rescale the transformed data
    scale = StandardScaler().fit(Xt_poly)     # find the desired transformation
    Xt_transformed = scale.transform(Xt_poly)  # & apply it to the training data
    Xe_transformed = scale.transform(Xe_poly)  # & apply it to the eval data

    # Fit your linear regression
    lr = learners[i]
    lr.fit(Xt_transformed, Yt)
    yt_pred = lr.predict(Xt_transformed)
    ye_pred = lr.predict(Xe_transformed)
    mse_train[i] = mse(Yt,yt_pred)
    mse_test[i] = mse(Ye,ye_pred)

plt.semilogy(degrees, mse_train, label='Training MSE', marker='o') # plot mse_train and mse_test as a function of t
plt.semilogy(degrees, mse_test, label='Evaluation MSE', marker='x') # plot mse_train and mse_test as a function of
plt.title('MSE as a function of Polynomial Degree')
plt.xlabel('Polynomial Degree')
plt.ylabel('Mean Squared Error (MSE)')
plt.legend()
plt.grid()
plt.show()
```



P3.3 Model Selection

Which degree would you select to use?

I will choose degree = 7 since it has the minimum MSE on the evaluation dataset.

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.

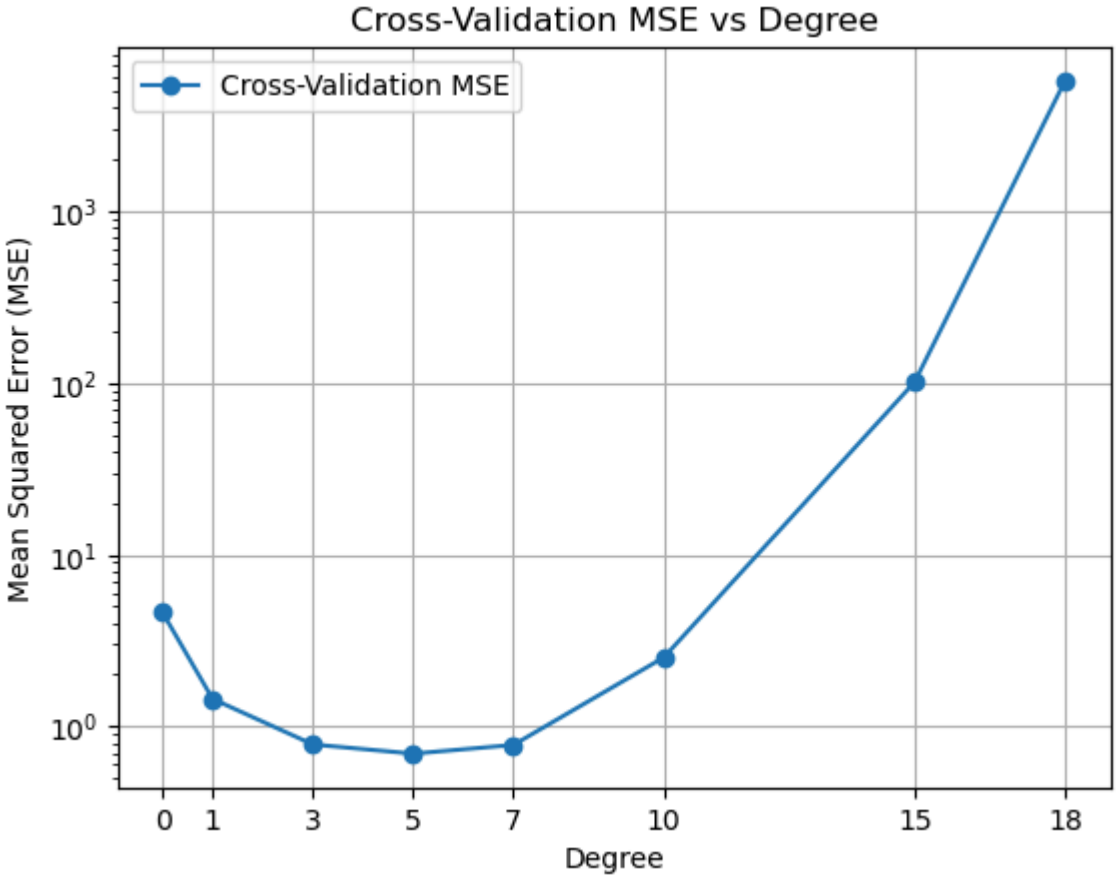
P4.1: 5-Fold Cross-validation

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:

In [53]:
```
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):
        # Extract the ith cross-validation fold (training/validation split)
        Xti,Xvi,Yti,Yvi = Xt[train_index],Xt[val_index],Yt[train_index],Yt[val_index]
        if degree == 0:
            m,n = Xti.shape
            Xti_poly = np.zeros((m,1))
            Xti_poly[:,0] = Xti[:,0]**0

            m,n = Xvi.shape
            Xvi_poly = np.zeros((m,1))
            Xvi_poly[:,0] = Xvi[:,0]**0
        else:
            # Now, build the model:
            # Create a polynomial feature expansion
```

```
            poly = PolynomialFeatures(degree=degree, include_bias=False).fit(Xti)
            Xti_poly = poly.fit_transform(Xti)  # Transform training data with polynomial features

            poly = PolynomialFeatures(degree=degree, include_bias=False).fit(Xvi)
            Xvi_poly = poly.transform(Xvi)  # Transform validation data with the same features

        # Create a StandardScaler
        scale = StandardScaler().fit(Xti_poly)
        Xti_scaled = scale.transform(Xti_poly)  # Standardize training data
        Xvi_scaled = scale.transform(Xvi_poly)  # Standardize validation data

        # Fit the linear regression model on the training folds, Xti/Yti
        lr = LinearRegression()
        lr.fit(Xti_scaled, Yti)

        # Compute the MSE on the evaluation fold, Xvi/Yvi
        y_pred = lr.predict(Xvi_scaled)  # Make predictions on validation set
        mse_fold.append(mse(Yvi, y_pred))  # Append the MSE for each fold

    # Evaluate the quality of this degree by averaging the MSE across the five folds
    mse_xval[j] = np.mean(mse_fold)  # Compute and store the average MSE

# Plot the estimated MSE from cross-validation as a function of the degree
plt.semilogy(degrees, mse_xval, 'o-', label='Cross-Validation MSE')  # Plot cross-validation MSE
plt.title('Cross-Validation MSE vs Degree')
plt.xlabel('Degree')
plt.ylabel('Mean Squared Error (MSE)')
plt.xticks(degrees)
plt.grid()
plt.legend()
plt.show()
    ###  YOUR CODE ENDS HERE  ###
```



P4.2: Cross-validation model selection

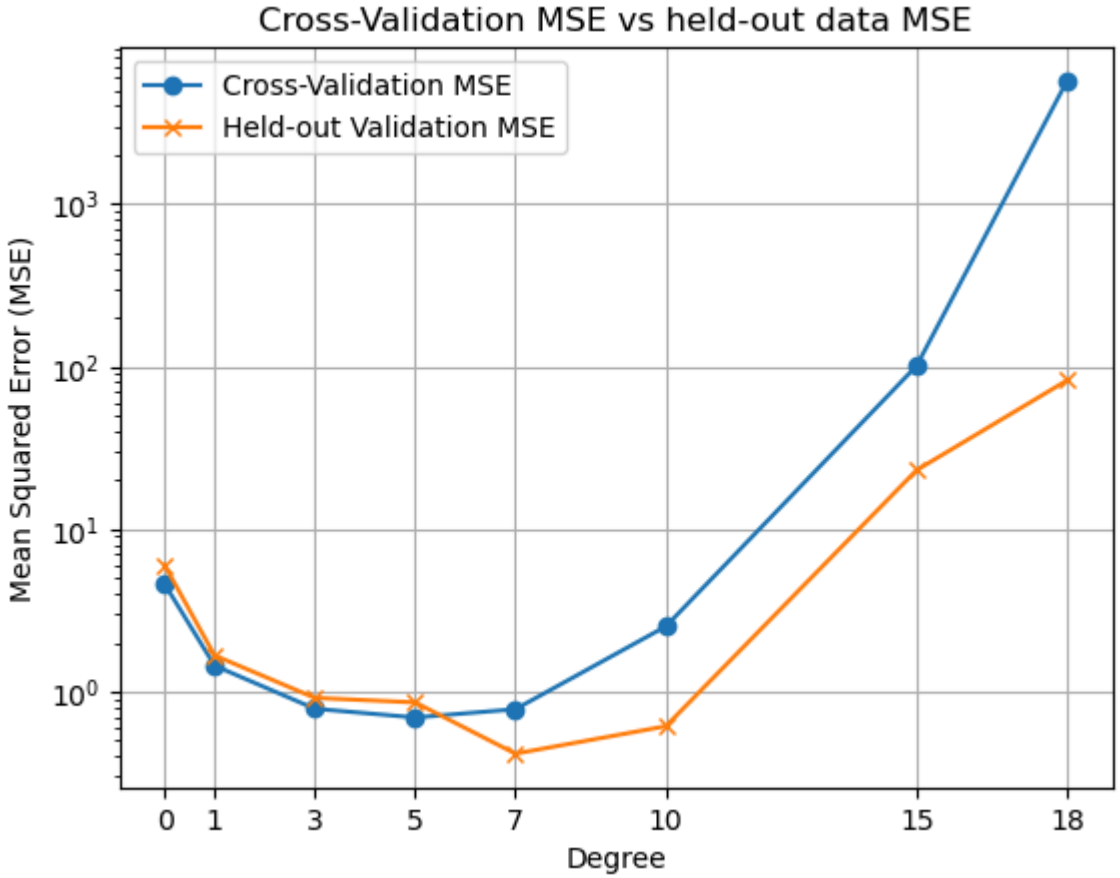
What degree would you choose based on the cross validation performance?

Based on the cross validation performance, I will choose degree = 5 since it has the minimum MSE.

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.

In [58]:
```
# Plot two MSE curves to compare
plt.semilogy(degrees, mse_xval, 'o-', label='Cross-Validation MSE')  # Plot cross-validation MSE
plt.semilogy(degrees, mse_test, label='Held-out Validation MSE', marker='x') # plot held-out evalaution data MSE
plt.title('Cross-Validation MSE vs held-out data MSE')
plt.xlabel('Degree')
plt.ylabel('Mean Squared Error (MSE)')
plt.xticks(degrees)
plt.grid()
plt.legend()
plt.show()
```



When degrees less than 7 (Underfitting), the model's complexity is insufficient to capture the underlying patterns in the data. This leads to lower cross-validation MSE. It indicates that the model is not fitting the data adequately. When degrees greater than 5, the model starts to overfit the training data. Its performance on unseen test data can decline because the model captures noise and specific details in the training data. This leads to a situation where the true MSE drops below the cross-validation MSE, indicating decreased generalization ability.

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.

P5.1: Regularized Regression

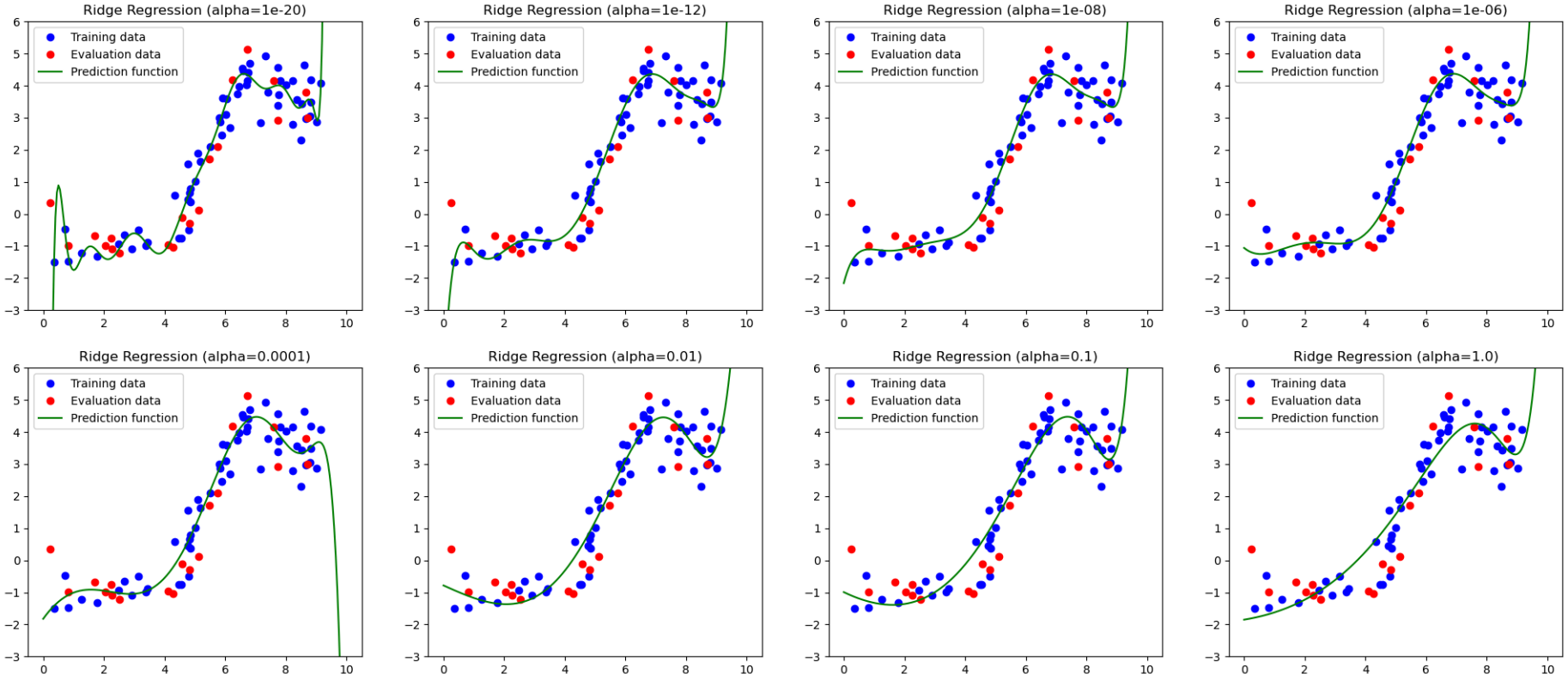
In `sklearn`, linear regression with quadratic (L2) regularization is implemented in a different object, `Ridge`. Use this ridge regression model to fit your degree-18 data using various amounts of regularization:

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

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

In [67]:
```
alphas = [1e-20, 1e-12, 1e-8, 1e-6, 1e-4, 1e-2, 1e-1, 1.]
learners = [ None ]*len(alphas)

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

for i,alpha in enumerate(alphas):

    ### YOUR CODE STARTS HERE ###
    pipe = Pipeline([
        ('poly', PolynomialFeatures(degree=18, include_bias=False)),  # Polynomial feature expansion
        ('scaler', StandardScaler()),  # Standard scaling
        ('ridge', Ridge(alpha=alpha))  # Ridge regression
    ])   # define your high-dim transform, scaling, and ridge regression learner

    # Fit your learner and save it to your list
    learners[i] = pipe.fit(Xt, Yt)
    x_spaced = np.linspace(0, 10, 200).reshape(-1, 1)
    y_pred = learners[i].predict(x_spaced)
    axi = ax[i//4,i%4]
    axi.plot(Xt, Yt, 'bo', label='Training data')  # plot the data and your prediction function
    axi.plot(Xe, Ye, 'ro', label='Evaluation data')  # Plot evaluation data
    axi.plot(x_spaced, y_pred, 'g-', label='Prediction function')  # Plot predictions
    axi.set_ylim([-3, 6])  # you'll want to set a consistent y-scale for comparison
    axi.set_title(f'Ridge Regression (alpha={alpha})')  # don't forget to label your plots
```

```
    axi.legend()  # Show legend
    ###  YOUR CODE ENDS HERE  ###
```

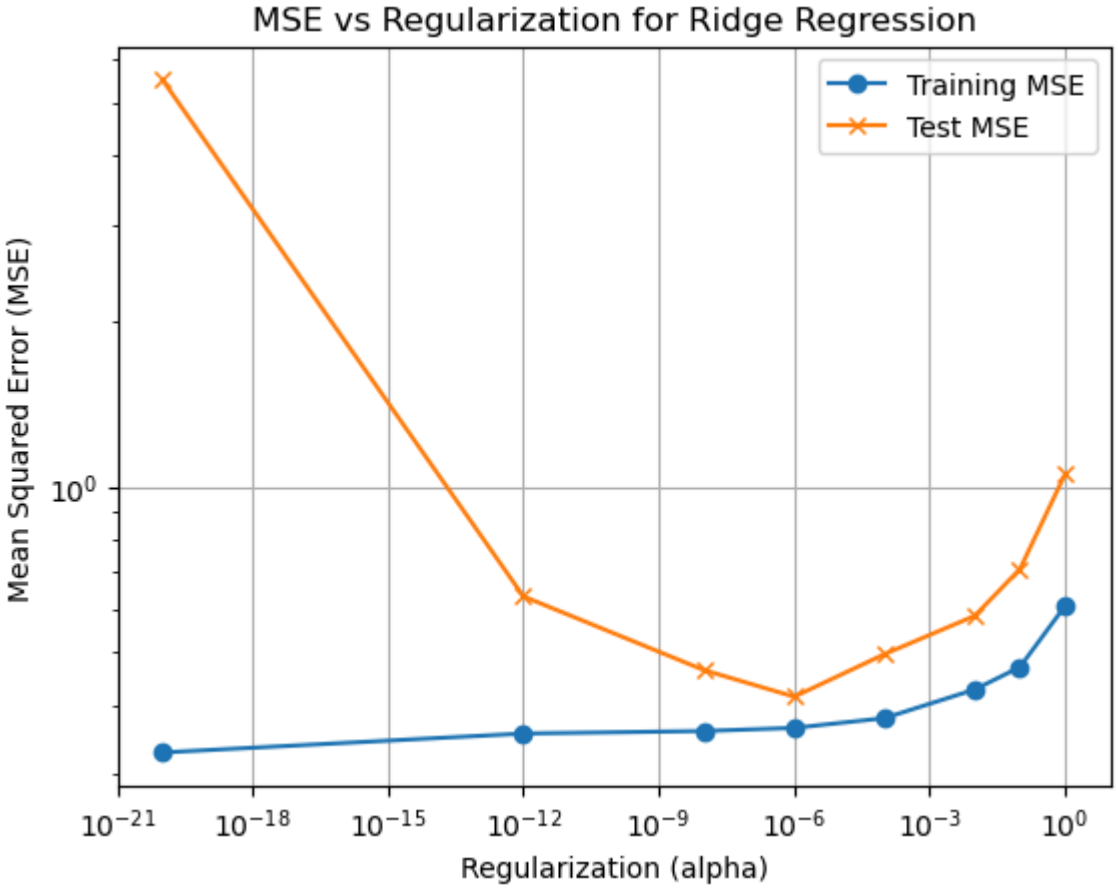


P5.2: Training and Test Performance

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.)

In [70]:
```
mse_train = [0]*len(alphas)
mse_test = [0]*len(alphas)

for i,alpha in enumerate(alphas):
    pipe = Pipeline([
        ('poly', PolynomialFeatures(degree=18, include_bias=False)),
```

```
        ('scaler', StandardScaler()),
        ('ridge', Ridge(alpha=alpha))
    ])

    # Fit the model on training data
    pipe.fit(Xt, Yt)

    # Predict on training data
    y_train_pred = pipe.predict(Xt)
    mse_train[i] = mse(Yt, y_train_pred)

    # Predict on evaluation data
    y_test_pred = pipe.predict(Xe)
    mse_test[i] = mse(Ye, y_test_pred)

# plot mse_train and mse_test as a function of the regularization
plt.loglog(alphas, mse_train, label='Training MSE', marker='o')
plt.loglog(alphas, mse_test, label='Test MSE', marker='x')
plt.xlabel('Regularization (alpha)')
plt.ylabel('Mean Squared Error (MSE)')
plt.title('MSE vs Regularization for Ridge Regression')
plt.legend()
plt.grid()
plt.show()
```



P5.3: Model Selection

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

I will select regularization value α = 10e-6 , since it has the minimum test MSE. When α < 10e-6, the training MSE is low, but the test MSE is significantly higher, indicating that the model is overfitting. When α > 10e-6, the MSEs for both training and test data may be high, indicating that the model is underfitting.

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.

This assignment was completed independently by me.