Short Answer and "True or False" Conceptual questions

A1. The answers to these questions should be answerable without referring to external materials. Briefly justify your answers with a few words.

a. *[2 points]* **In your own words, describe what bias and variance are. What is the bias-variance tradeoff?**

b. *[2 points]* **What *typically* happens to bias and variance when the model complexity increases/decreases?**

c. *[2 points]* **True or False: Suppose you're given a fixed learning algorithm. If you collect more training data from the same distribution, the variance of your predictor increases.**

d. *[2 points]* **Suppose that we are given train, validation, and test sets. Which of these sets should be used for hyperparameter tuning? Explain your choice and detail a procedure for hyperparameter tuning.**

e. *[1 point]* **True or False: The training error of a function on the training set provides an overestimate of the true error of that function.**

What to Submit:

- **Parts c, e:** True or False
- **Parts a-e:** Brief (2-3 sentence) explanation justifying your answer.

Part a:

Bias is the difference between the real data (value) and the model predicted data (value). A high bias is a high degree of error between the training and test data. If you had a high bias it probably means that the model does not follow the training data closely. A High bias is like shooting off target. You get a gigh bias by having simplified assumptions.

Variance is how much the model will vary for a value. High variance is like when shooting a target and it is hitting all over the target while a low variance is like shooting a very tightly grouped set of shots. A high variance willl come by having a model pay a lot of attention to the training data. High variance can perform well on training data but not nearly as well on the tesign or "real values"

The Bias-variance tradeoff is one that bias and variance are generally inverse of the other. Both a high variance and a high bias cause us to have bad models. high variance causes us to overfit our models to the training data while a high bias causes us to underfit our models to the data. What we are hunting for is the Goldilocks zone (prefferabbly a low bias and low variance). Really what it coomes down to is finding the right balance between vairance and bias when we train our data to minimize errors

Part b:

as complexity goes up: variance goes up; bias goes down as complexity goes down: variance goes down, bias goes up

Part c:

False.

If you are training an FLA and are collecting more training from the SAME distribution, you are going to cause variance to go down. Basiacally the model is going to be better attuned to the training data. However this may cause overfitting of the model.

Part d:

If given the three sets of data to hyperparameter train we should use the training data to train the model as the first step of hypervalidation. Then we shouldd use the validation data as a way to tune our hyperparamter model. We should NOT use the test set for trainign our model. EVER.

A brief procedure:

1. Set the "hyperparameters" as best you can using knowledge already obtained
2. Test with Validation set
3. Adjust the hyperparameters
4. Test again with validation set
5. Wash, rinse, repeat until the hyperparameters yield best performance on validation
6. Retrain the model with "best" hyperparameters using botht the traingin and validation sets.
7. Evaluate with test set of data

Part e:

False

It will provide an underestimate of the true error of the function. That is because the model has been trained on the training data set and the trainign error is the error measured with the trainign data. The model is going to perform BEST with the training data so the REAL error will be higher. Hopefully not too muchb higher though.

A2. You're the Reign FC manager, and the team is five games into its 2021 season. The numbers of goals scored by the team in each game so far are given below:

$$[3, 7, 5, 0, 2].$$

Let's call these scores $x_1, \ldots, x_5$. Based on your (assumed iid) data, you'd like to build a model to understand how many goals the Reign are likely to score in their next game. You decide to model the number of goals scored per game using a *Poisson distribution*. Recall that the Poisson distribution with parameter λ assigns every non-negative integer $x = 0, 1, 2, \ldots$ a probability given by

$$\text{Poi}(x|\lambda) = e^{-\lambda}\frac{\lambda^x}{x!}.$$

a. *[5 points]* **Derive an expression for the maximum-likelihood estimate of the parameter λ governing the Poisson distribution in terms of goal counts for the first n games: $x_1, \ldots, x_n$. (Hint: remember that the log of the likelihood has the same maximizer as the likelihood function itself.)**

b. *[2 points]* **Give a numerical estimate of λ after the first five games. Given this λ, what is the probability that the Reign score exactly 4 goals in their next game?**

c. *[2 points]* **Suppose the Reign actually score 8 goals in their 6th game. Give an updated numerical estimate of λ after six games and compute the probability that the Reign score exactly 5 goals in their 7th game.**

What to Submit:

- **Part a:** An expression for the MLE of λ after n games and relevant derivation
- **Part b:** A numerical estimate for λ and the probability that the Reign score 4 goals in their sixth game
- **Part c:** A numerical estimate for λ and the probability that the Reign score 5 goals in their seventh game

part a:

given:

$$Poi(x \vee \lambda) = \frac{e^{-\lambda}\lambda^x}{x!}$$

the liklihood functions for x1,x2,....xi:

$$L(\lambda) = \prod_{i=1}^{n} \frac{e^{-\lambda}\lambda^{x_i}}{x_i!}$$

(Hint: remember that the log of the likelihood has the same maximizer as the likelihood function itself)

$$log(L(\lambda)) = log\left(\prod_{i=1}^{n} \frac{e^{-\lambda}\lambda^{x_i}}{x_i!}\right) = \sum_{i=1}^{n} log(e^{-\lambda}) + log(\lambda^{x_i}) - log(x_i!) = \sum_{i=1}^{n} \left(-\lambda + x_i log(\lambda) - log(x!)\right)$$

So:

$$log(L(\lambda)) = \sum_{i=1}^{n} \left(-\lambda + x_i log(\lambda) - log(x!)\right) = -\lambda n + \sum_{i=1}^{n} \left(x_i log(\lambda) - log(x!)\right)$$

Set the derivative of $log(L(\lambda)) = 0$

$$\frac{d\,log(L(\lambda))}{dx}=0=\frac{d}{dx}\left(-\lambda n+\sum_{i=1}^{n}\left(x_i log(\lambda)-log(x!)\right)\right)=-n+\sum_{i=1}^{n}\frac{x_i}{\lambda}=0$$

Rearrange:

$$\lambda=\sum_{i=1}^{n}\frac{x_i}{n}$$

.

Part b:

$$\lambda=\sum_{i=1}^{n}\frac{x_i}{n}=\sum_{i=1}^{5}\frac{x_i}{5}=\frac{1}{5}(3+7+5+0+2)=3.4=\frac{17}{5}$$

$$Poi\left(4\vee\frac{17}{5}\right)=\frac{e^{-\frac{17}{5}}\frac{17}{5}^4}{4!}=0.185824592$$

part c:

$$\lambda=\sum_{i=1}^{n}\frac{x_i}{n}=\sum_{i=1}^{5}\frac{x_i}{6}=\frac{1}{6}(3+7+5+0+2+8)=4.16666667=\frac{25}{6}$$

$$Poi\left(5\vee\frac{25}{6}\right)=\frac{e^{-\frac{25}{6}}\frac{25}{6}^5}{5!}=0.162256538$$

Polynomial Regression

Relevant Files[1]:

- **polyreg.py**
- `linreg_closedform.py`
- `plot_polyreg_univariate.py`
- `plot_polyreg_learningCurve.py`

A3. Recall that polynomial regression learns a function $h_{\boldsymbol{\theta}}(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \ldots + \theta_d x^d$, where d represents the polynomial's highest degree. We can equivalently write this in the form of a linear model with d features

$$h_{\boldsymbol{\theta}}(x) = \theta_0 + \theta_1 \phi_1(x) + \theta_2 \phi_2(x) + \ldots + \theta_d \phi_d(x) \ , \tag{1}$$

using the basis expansion that $\phi_j(x) = x^j$. Notice that, with this basis expansion, we obtain a linear model where the features are various powers of the single univariate x. We're still solving a linear regression problem, but are fitting a polynomial function of the input.

a. *[8 points]* **Implement regularized polynomial regression in `polyreg.py`.** You may implement it however you like, using gradient descent or a closed-form solution. However, we would recommend the closed-form solution since the data sets are small; for this reason, we've included an example closed-form implementation of regularized linear regression in `linreg_closedform.py` (you are welcome to build upon this implementation, but make CERTAIN you understand it, since you'll need to change several lines of it). Note that all matrices are 2D NumPy arrays in the implementation.

   - `__init__(degree=1, regLambda=1E-8)` : constructor with arguments of d and λ
   - `fit(X,Y)`: method to train the polynomial regression model
   - `predict(X)`: method to use the trained polynomial regression model for prediction
   - `polyfeatures(X, degree)`: expands the given $n \times 1$ matrix X into an $n \times d$ matrix of polynomial features of degree d. Note that the returned matrix will not include the zero-th power.

Note that the `polyfeatures(X, degree)` function maps the original univariate data into its higher order powers. Specifically, X will be an $n \times 1$ matrix ($X \in \mathbb{R}^{n \times 1}$) and this function will return the polynomial expansion of this data, a $n \times d$ matrix. Note that this function will **not** add in the zero-th order feature (i.e., $x_0 = 1$). You should add the x_0 feature separately, outside of this function, before training the model.

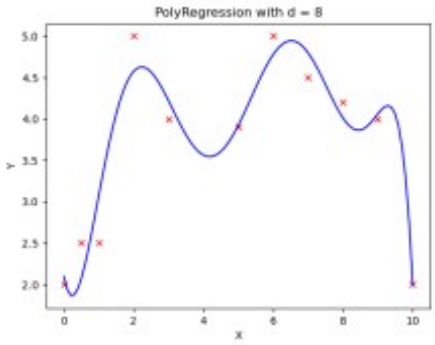


Figure 2: Fit of polynomial regression with $\lambda = 0$ and $d = 8$

By not including the x_0 column in the matrix `polyfeatures()`, this allows the `polyfeatures` function to be more general, so it could be applied to multi-variate data as well. (If it did add the x_0 feature, we'd end up with multiple columns of 1's for multivariate data.)

Also, notice that the resulting features will be badly scaled if we use them in raw form. For example, with a polynomial of degree $d = 8$ and $x = 20$, the basis expansion yields $x^1 = 20$ while $x^8 = 2.56 \times 10^{10}$ –

[1] **Bold text** indicates files or functions that you will need to complete; you should not need to modify any of the other files.

an absolutely huge difference in range. Consequently, we will need to standardize the data before solving linear regression. Standardize the data in `fit()` after you perform the polynomial feature expansion. You'll need to apply the same standardization transformation in `predict()` before you apply it to new data.

b. *[2 points]* **Run `plot_polyreg_univariate.py` to test your implementation, which will plot the learned function.** In this case, the script fits a polynomial of degree $d = 8$ with no regularization $\lambda = 0$. From the plot, we see that the function fits the data well, but will not generalize well to new data points. Try increasing the amount of regularization, and in 1-2 sentences, describe the resulting effect on the function (you may also provide an additional plot to support your analysis).

What to Submit:

- **Part a: Code** on Gradescope through coding submission.
- **Part b:** 1-2 sentence description of the effect of increasing regularization.
- **Part b:** Plots before and after increase in regularization.

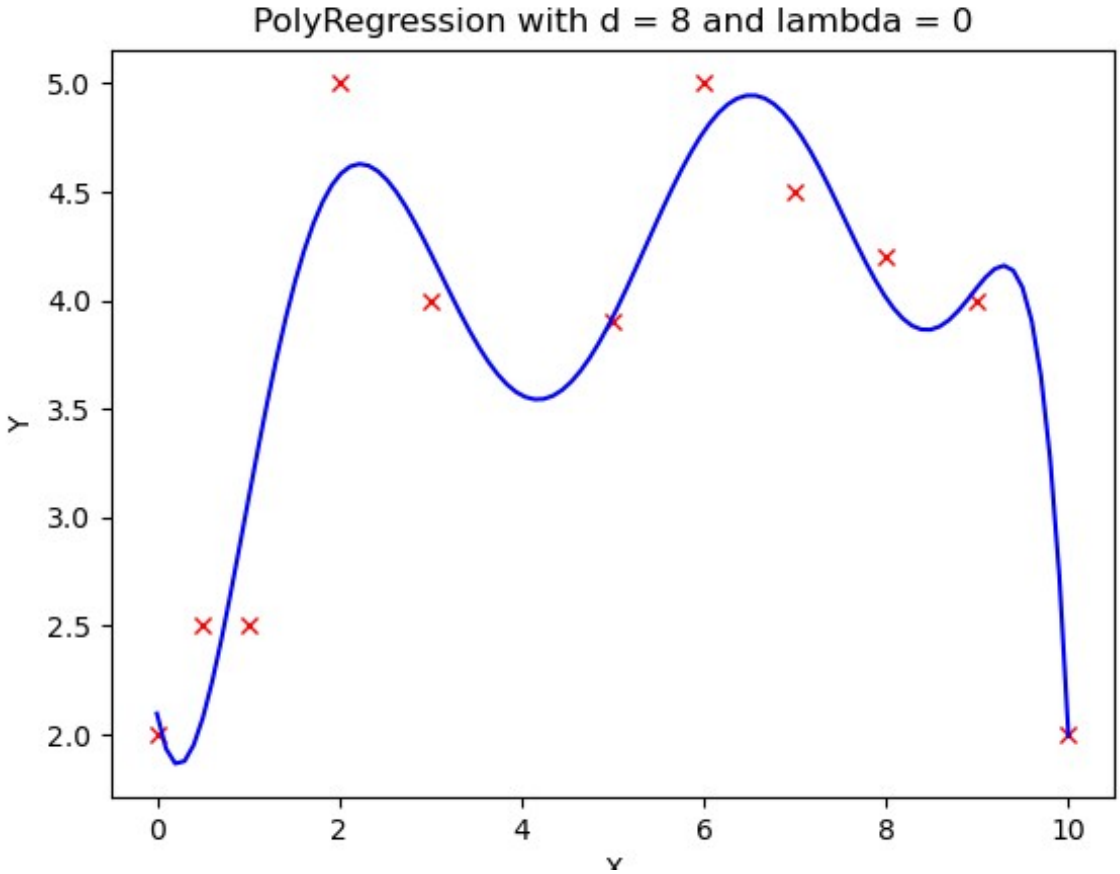
Part a Code:

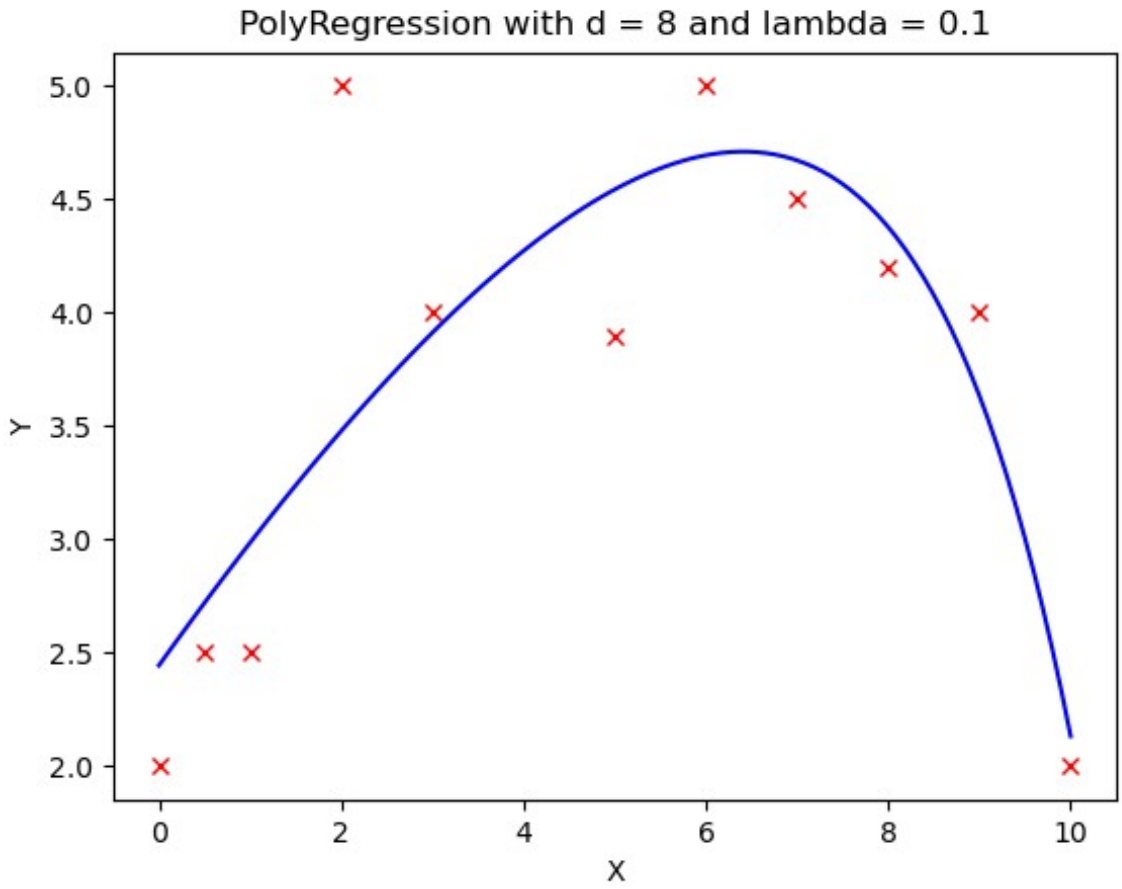
```
"""
    Template for polynomial regression
    AUTHOR Eric Eaton, Xiaoxiang Hu
"""

from typing import Tuple

import numpy as np

from utils import problem


class PolynomialRegression:
    @problem.tag("hw1-A", start_line=5)
    def __init__(self, degree: int = 1, reg_lambda: float = 1e-8):
        """Constructor
        """
        self.degree: int = degree
        self.reg_lambda: float = reg_lambda
        # Fill in with matrix with the correct shape
        self.weight: np.ndarray = None  # type: ignore
        # You can add additional fields
        # Added fields mean and std for def fit
        self.mean = None
        self.std = None

    @staticmethod
    @problem.tag("hw1-A")
    def polyfeatures(X: np.ndarray, degree: int) -> np.ndarray:
        """
        Expands the given X into an (n, degree) array of polynomial
```

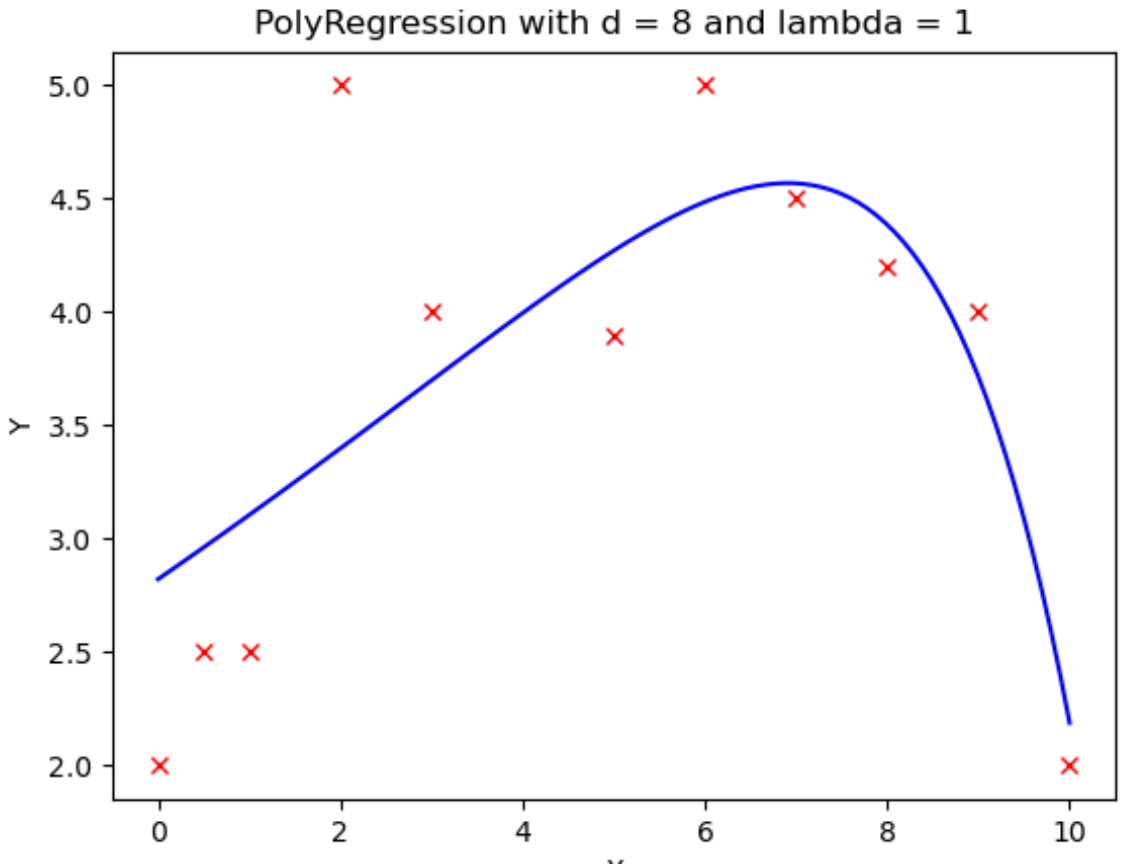
```
features of degree degree.

        Args:
            X (np.ndarray): Array of shape (n, 1).
            degree (int): Positive integer defining maximum power to
include.

        Returns:
            np.ndarray: A (n, degree) numpy array, with each row
comprising of
                X, X * X, X ** 3, ... up to the degree^th power of X.
                Note that the returned matrix will not include the
zero-th power.

        """
        # find the size of n
        n = len(X)
        #create array of incrementing ints up to the degree
        exp = np.arange(1,degree+1)
        # calculate the polynomial feature and return
        polyX = np.hstack([X**d for d in exp])
        return polyX


    @problem.tag("hw1-A")
    def fit(self, X: np.ndarray, y: np.ndarray):
        """
        Trains the model, and saves learned weight in self.weight

        Args:
            X (np.ndarray): Array of shape (n, 1) with observations.
            y (np.ndarray): Array of shape (n, 1) with targets.

        Note:
            You will need to apply polynomial expansion and data
standardization first.
        """
        #use the polyfeatures function
        polyX = self.polyfeatures(X,self.degree)
        #calculate mean and standard for each column across all rows
        self.mean = np.mean(polyX, axis = 0)
        self.std = np.std(polyX, axis = 0)
        #standardize
        polyX = (polyX-self.mean)/self.std
        #add intercept term by stacking a column of ones with polyX
that has the same number of rows as polyX
        polyX = np.c_[np.ones([polyX.shape[0]]), polyX]
        #make identity matrix with same number of columns
        mat_id = np.eye(polyX.shape[1])
        #set the first element to zero so the the intercept is not
```

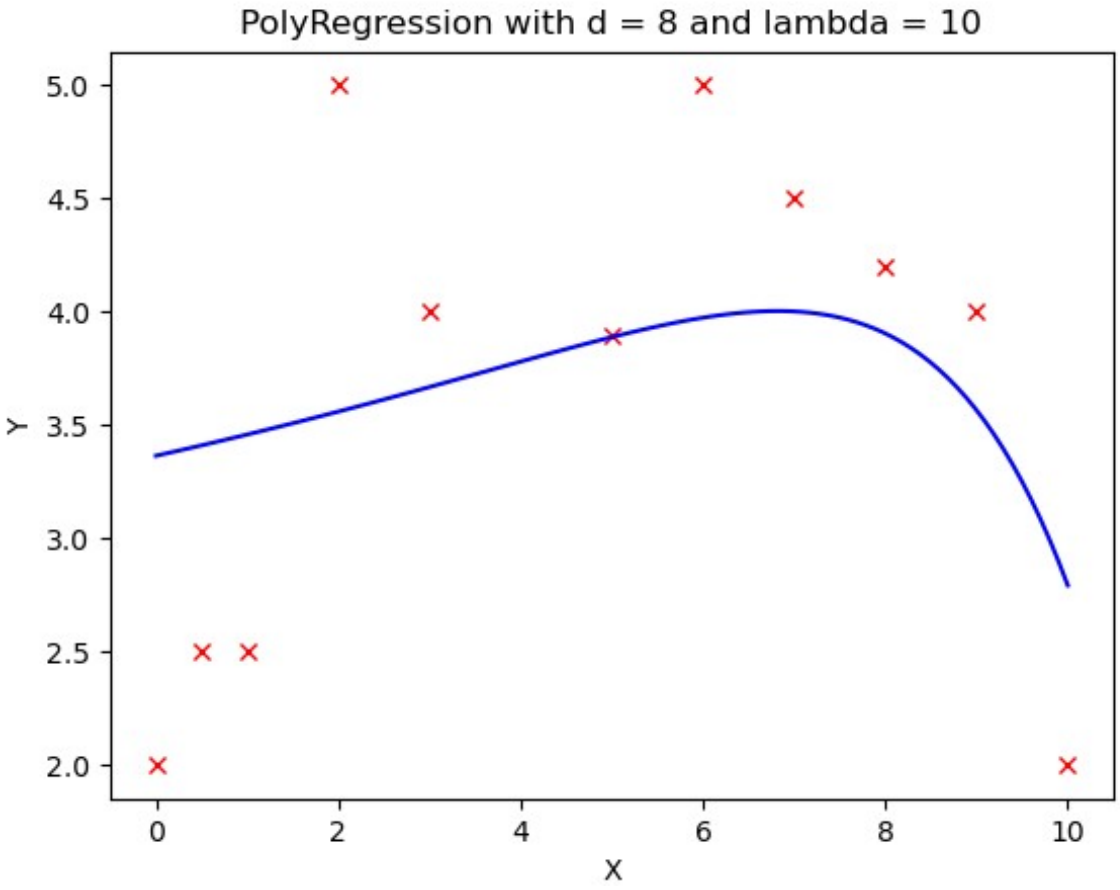
```
regularized
        mat_id[0,0] = 0
        #set and store the weight
        self.weight = np.linalg.pinv(polyX.T @ polyX + self.reg_lambda
* mat_id) @ polyX.T @ y

    @problem.tag("hw1-A")
    def predict(self, X: np.ndarray) -> np.ndarray:
        """
        Use the trained model to predict values for each instance in
X.

        Args:
            X (np.ndarray): Array of shape (n, 1) with observations.

        Returns:
            np.ndarray: Array of shape (n, 1) with predictions.
        """
        #use the polyfeatures function
        polyX = self.polyfeatures(X, self.degree)
        #standardize the data
        polyX = (polyX-self.mean)/self.std
        #add the intercept term
        polyX = np.c_[np.ones(polyX.shape[0]), polyX]
        #return the predicted values
        return polyX @ self.weight


@problem.tag("hw1-A")
def mean_squared_error(a: np.ndarray, b: np.ndarray) -> float:
    """Given two arrays: a and b, both of shape (n, 1) calculate a
mean squared error.

    Args:
        a (np.ndarray): Array of shape (n, 1)
        b (np.ndarray): Array of shape (n, 1)

    Returns:
        float: mean squared error between a and b.
    """
    return ((a-b)**2).mean()
```

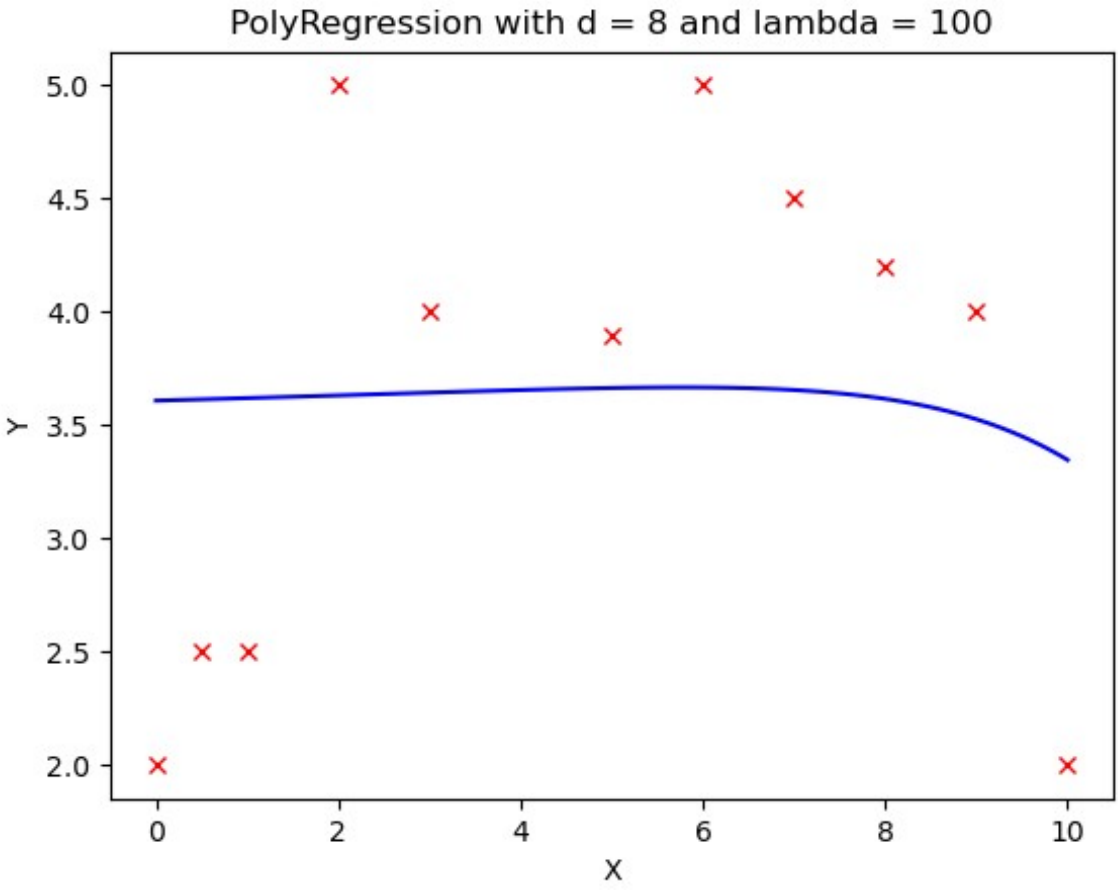
Part b plots:

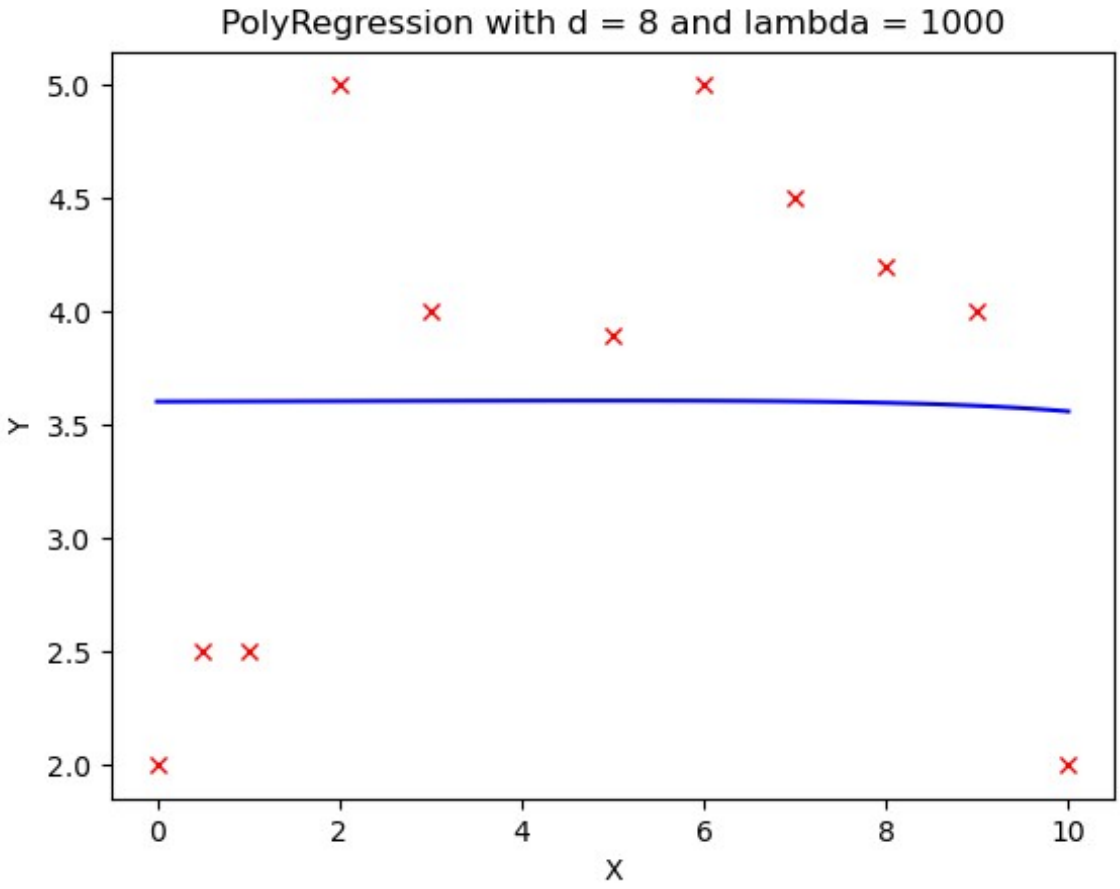

PolyRegression with d = 8 and lambda = 0
5.0
4.5
4.0
3.5
3.0
2.5
2.0
Y
0
2
4
6
8
10
X

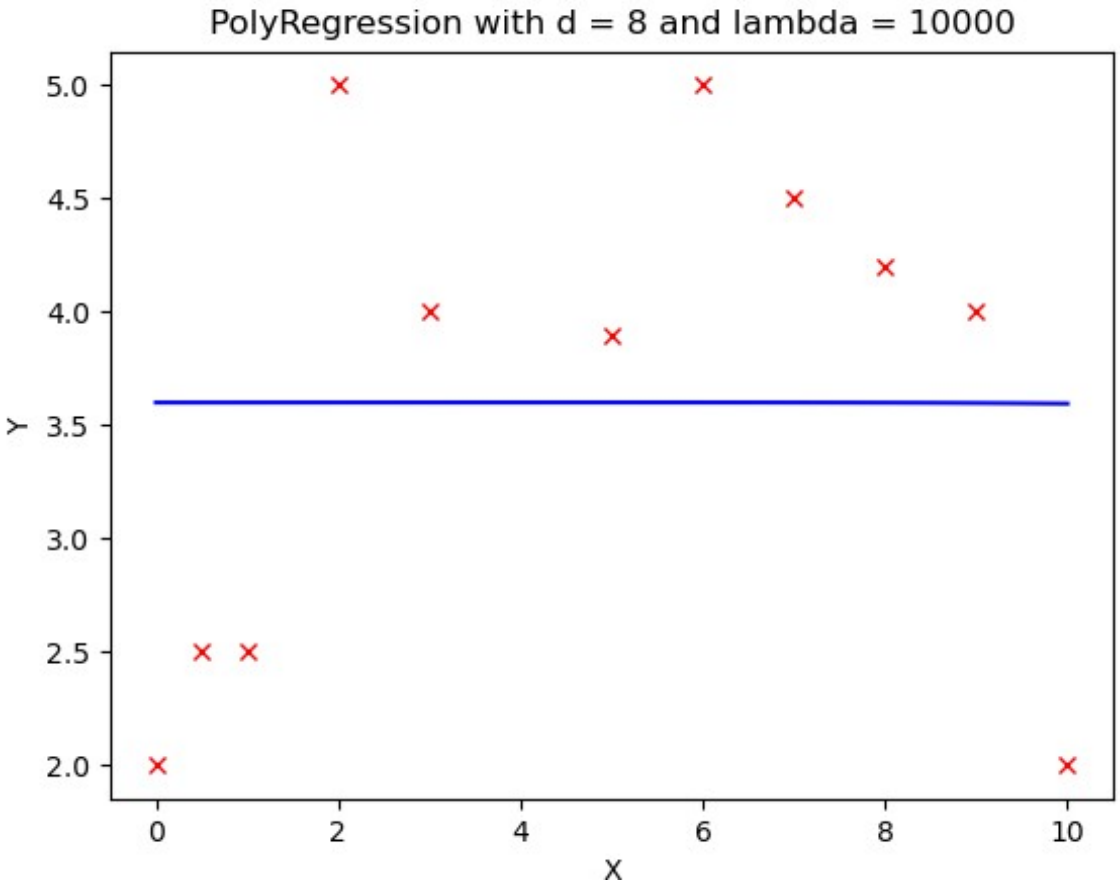

PolyRegression with d = 8 and lambda = 0.1
5.0
4.5
4.0
3.5
3.0
2.5
2.0
Y
0
2
4
6
8
10
X


PolyRegression with d = 8 and lambda = 1
5.0
4.5
4.0
3.5
3.0
2.5
2.0
Y
0
2
4
6
8
10


PolyRegression with d = 8 and lambda = 10
5.0
4.5
4.0
3.5
3.0
2.5
2.0
Y
0
2
4
6
8
10
X


PolyRegression with d = 8 and lambda = 100
5.0
4.5
4.0
3.5
3.0
2.5
2.0
Y
0
2
4
6
8
10
X


PolyRegression with d = 8 and lambda = 1000
5.0
4.5
4.0
3.5
3.0
2.5
2.0
Y
0
2
4
6
8
10
X



Part b description:

As the regularization increases we see that our models variance decreases and its bias increases. The higher the regularization the weaker our model was as we were underfitting the data. By increasing the regularization we are minimizing the Ordinary Least Squares because the sum of squares decreases. In the case above with regularization at 0 we risk overfitting. We would need to find the right balance of regularization so that the model would fit new data well not just the training data.

A4. *[10 points]* In this problem we will examine the bias-variance tradeoff through learning curves. Learning curves provide a valuable mechanism for evaluating the bias-variance tradeoff.

Implement the `learningCurve()` function in `polyreg.py` to compute the learning curves for a given training/test set. The `learningCurve(Xtrain, ytrain, Xtest, ytest, degree, regLambda)` function should take in the training data (`Xtrain`, `ytrain`), the testing data (`Xtest`, `ytest`), and values for the polynomial degree d and regularization parameter λ. The function should return two arrays, `errorTrain` (the array of training errors) and `errorTest` (the array of testing errors). The i^{th} index (start from 0) of each array should return the training error (or testing error) for learning with $i+1$ training instances. Note that the 0^{th} index actually won't matter, since we typically start displaying the learning curves with two or more instances.

When computing the learning curves, you should learn on `Xtrain`$[0{:}i]$ for $i = 1, \ldots,$ numInstances(`Xtrain`), each time computing the testing error over the **entire** test set. There is no need to shuffle the training data, or to average the error over multiple trials – just produce the learning curves for the given training/testing sets with the instances in their given order. Recall that the error for regression problems is given by

$$\frac{1}{n}\sum_{i=1}^{n}(h_{\boldsymbol{\theta}}(\mathbf{x}_i) - y_i)^2 \ . \tag{2}$$

Once the function is written to compute the learning curves, run the `plot_polyreg_learningCurve.py` script to plot the learning curves for various values of λ and d. You should see plots similar to the following:

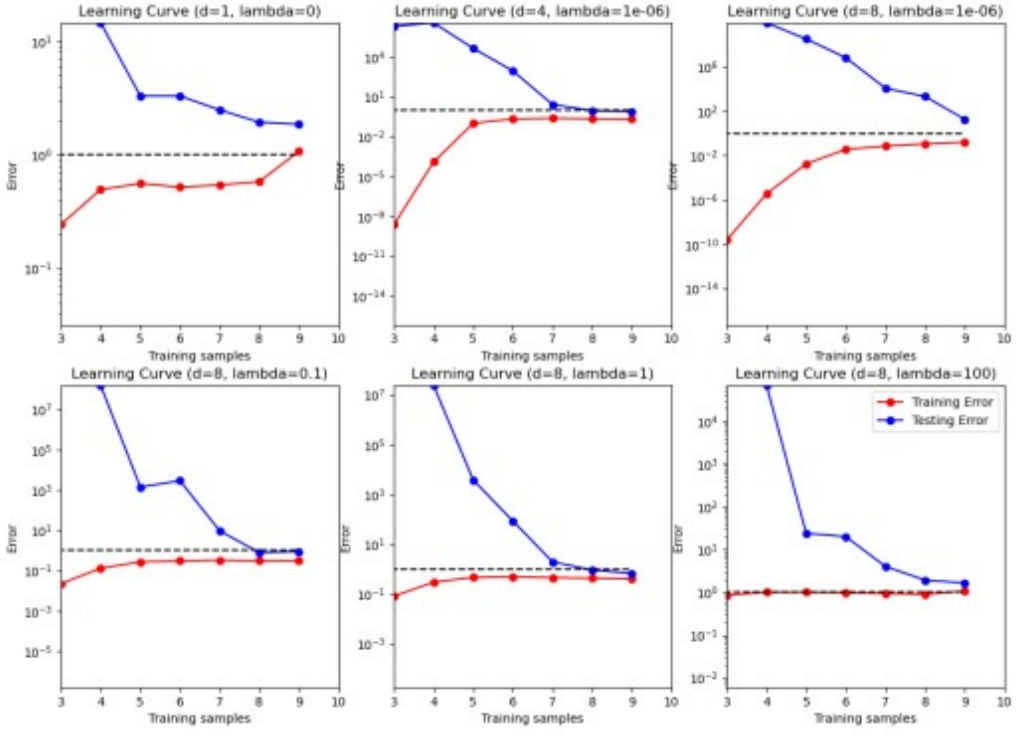


Figure 3: Learning curves for various values of d and λ. The blue lines represent the testing error, while the red lines the training error.

Notice the following:

- The y-axis is using a log-scale and the ranges of the y-scale are all different for the plots. The dashed black line indicates the $y = 1$ line as a point of reference between the plots.
- The plot of the unregularized model with $d = 1$ shows poor training error, indicating a high bias (i.e., it is a standard univariate linear regression fit).
- The plot of the (almost) unregularized model ($\lambda = 10^{-6}$) with $d = 8$ shows that the training error is low, but that the testing error is high. There is a huge gap between the training and testing errors caused by the model overfitting the training data, indicating a high variance problem.
- As the regularization parameter increases (e.g., $\lambda = 1$) with $d = 8$, we see that the gap between the training and testing error narrows, with both the training and testing errors converging to a low value. We can see that the model fits the data well and generalizes well, and therefore does not have either a high bias or a high variance problem. Effectively, it has a good tradeoff between bias and variance.
- Once the regularization parameter is too high ($\lambda = 100$), we see that the training and testing errors are once again high, indicating a poor fit. Effectively, there is too much regularization, resulting in high bias.

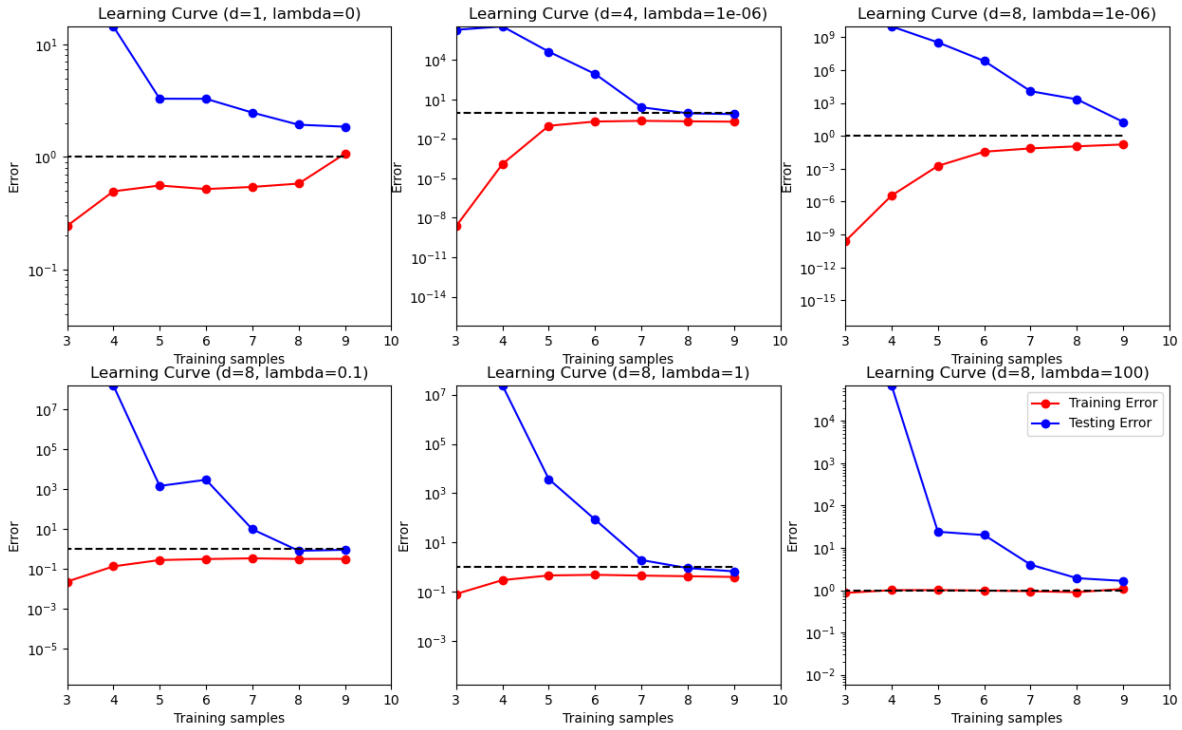
Submit plots for the same values of d and λ shown here. Make absolutely certain that you understand these observations, and how they relate to the learning curve plots. In practice, we can choose the value for λ via cross-validation to achieve the best bias-variance tradeoff.

Note: your learning curves slightly differ from ones above depending on whether you use 'np.linalg.solve' or directly implement the closed-form solution. Both solutions are correct.

What to Submit:

- **Plots** (or single plot with many subplots) of learning curves for $(d, \lambda) \in \{(1, 0), (4, 10^{-6}), (8, 10^{-6}), (8, 0.1), (8, 1), (8, 100)\}$.
- **Code** on Gradescope through coding submission

Plots:



Code:

```python
@problem.tag("hw1-A", start_line=5)
def learningCurve(
    Xtrain: np.ndarray,
    Ytrain: np.ndarray,
    Xtest: np.ndarray,
    Ytest: np.ndarray,
    reg_lambda: float,
    degree: int,
) -> Tuple[np.ndarray, np.ndarray]:
    """Compute learning curves.

    Args:
        Xtrain (np.ndarray): Training observations, shape: (n, 1)
        Ytrain (np.ndarray): Training targets, shape: (n, 1)
        Xtest (np.ndarray): Testing observations, shape: (n, 1)
        Ytest (np.ndarray): Testing targets, shape: (n, 1)
        reg_lambda (float): Regularization factor
        degree (int): Polynomial degree

    Returns:
        Tuple[np.ndarray, np.ndarray]: Tuple containing:
            1. errorTrain -- errorTrain[i] is the training mean
```

```
squared error using model trained by Xtrain[0:(i+1)]
            2. errorTest -- errorTest[i] is the testing mean squared
error using model trained by Xtrain[0:(i+1)]

    Note:
        - For errorTrain[i] only calculate error on Xtrain[0:(i+1)],
since this is the data used for training.
            THIS DOES NOT APPLY TO errorTest.
        - errorTrain[0:1] and errorTest[0:1] won't actually matter,
since we start displaying the learning curve at n = 2 (or higher)
    """
    n = len(Xtrain)

    errorTrain = np.zeros(n)
    errorTest = np.zeros(n)
    # Fill in errorTrain and errorTest arrays
    for i in range (1,n):
        model = PolynomialRegression(degree, reg_lambda)
        model.fit(Xtrain[:i+1], Ytrain[:i+1])
        errorTrain[i] =
mean_squared_error(model.predict(Xtrain[:i+1]),Ytrain[:i+1])
        errorTest[i] = mean_squared_error(model.predict(Xtest), Ytest)
    return errorTrain, errorTest
```

Ridge Regression on MNIST

Relevant Files (you should not need to modify any of the other files for this part):

- **ridge_regression.py**

A5. In this problem, we will implement a regularized least squares classifier for the MNIST data set. The task is to classify handwritten images of numbers between 0 to 9.

You are **NOT** allowed to use any of the pre-built classifiers in `sklearn`. Feel free to use any method from `numpy` or `scipy`. **Remember:** if you are inverting a matrix in your code, you are probably doing something wrong (Hint: look at `scipy.linalg.solve`).

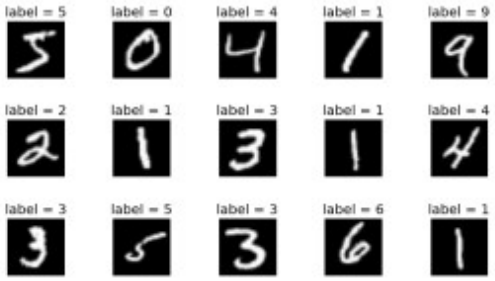


Figure 4: Sample images from the MNIST data set.

Each example has features $x_i \in \mathbb{R}^d$ (with $d = 28 * 28 = 784$) and label $z_j \in \{0, \ldots, 9\}$. **Since images are represented as 784-dimensional vectors, you can visualize a single example x_i with `imshow` after reshaping it to its original 28×28 image shape** (and noting that the label z_j is accurate). Checkout figure 4 for some sample images. We wish to learn a predictor $\widehat{f}$ that takes as input a vector in $\mathbb{R}^d$ and outputs an index in $\{0, \ldots, 9\}$. We define our training and testing classification error on a predictor f as

$$\widehat{\epsilon}_{\text{train}}(f) = \frac{1}{N_{\text{train}}} \sum_{(x,z) \in \text{Training Set}} \mathbf{1}\{f(x) \neq z\}$$

$$\widehat{\epsilon}_{\text{test}}(f) = \frac{1}{N_{\text{test}}} \sum_{(x,z) \in \text{Test Set}} \mathbf{1}\{f(x) \neq z\}$$

We will use one-hot encoding of the labels: for each observation (x, z), the original label $z \in \{0, \ldots, 9\}$ is mapped to the standard basis vector e_{z+1} where e_i is a vector of size k containing all zeros except for a 1 in the i^{th} position (positions in these vectors are indexed starting at one, hence the $z + 1$ offset for the digit labels). We adopt the notation where we have n data points in our training objective with features $x_i \in \mathbb{R}^d$ and label one-hot encoded as $y_i \in \{0, 1\}^k$. Here, $k = 10$ since there are 10 digits.

a. *[10 points]* In this problem we will choose a linear classifier to minimize the regularized least squares objective:

$$\widehat{W} = \text{argmin}_{W \in \mathbb{R}^{d \times k}} \sum_{i=1}^{n} \|W^T x_i - y_i\|_2^2 + \lambda \|W\|_F^2$$

Note that $\|W\|_F$ corresponds to the Frobenius norm of W, i.e. $\|W\|_F^2 = \sum_{i=1}^{d} \sum_{j=1}^{k} W_{i,j}^2$. To classify a

point x_i we will use the rule $\arg\max_{j=0,\ldots,9} e_{j+1}^T \widehat{W}^T x_i$. Note that if $W = \begin{bmatrix} w_1 & \ldots & w_k \end{bmatrix}$ then

$$\begin{aligned}\sum_{i=1}^{n} \|W^T x_i - y_i\|_2^2 + \lambda \|W\|_F^2 &= \sum_{j=1}^{k} \left[\sum_{i=1}^{n} (e_j^T W^T x_i - e_j^T y_i)^2 + \lambda \|W e_j\|_2^2 \right] \\ &= \sum_{j=1}^{k} \left[\sum_{i=1}^{n} (w_j^T x_i - e_j^T y_i)^2 + \lambda \|w_j\|_2^2 \right] \\ &= \sum_{j=1}^{k} \left[\|X w_j - Y e_j\|_2^2 + \lambda \|w_j\|_2^2 \right]\end{aligned}$$

where $X = \begin{bmatrix} x_1 & \ldots & x_n \end{bmatrix}^\top \in \mathbb{R}^{n \times d}$ and $Y = \begin{bmatrix} y_1 & \ldots & y_n \end{bmatrix}^\top \in \mathbb{R}^{n \times k}$. **Show that**

$$\widehat{W} = (X^T X + \lambda I)^{-1} X^T Y$$

b. *[9 points]*

- Implement a function `train` that takes as input $X \in \mathbb{R}^{n \times d}$, $Y \in \{0,1\}^{n \times k}$, $\lambda > 0$ and returns $\widehat{W} \in \mathbb{R}^{d \times k}$.
- Implement a function `one_hot` that takes as input $Y \in \{0, \ldots, k-1\}^n$, and returns $Y \in \{0,1\}^{n \times k}$.
- Implement a function `predict` that takes as input $W \in \mathbb{R}^{d \times k}$, $X' \in \mathbb{R}^{m \times d}$ and returns an m-length vector with the ith entry equal to $\arg\max_{j=0,\ldots,9} e_j^T W^T x_i'$ where $x_i' \in \mathbb{R}^d$ is a column vector representing the ith example from X'.
- Using the functions you coded above, train a model to estimate $\widehat{W}$ on the MNIST training data with $\lambda = 10^{-4}$, and make label predictions on the test data. This behavior is implemented in the `main` function provided in a zip file.

c. *[1 point]* **What are the training and testing errors of the classifier trained as above?**

d. *[2 points]* **Using matplotlib's `imshow` function, plot any 10 samples from the test data whose labels are incorrectly predicted by the classifier. Notice any patterns?**

Once you finish this problem question, you should have a powerful handwritten digit classifier! Curious to know how it compares to other models, including the almighty *Neural Networks*? Check out the **linear classifier (1-layer NN)** on the official MNIST leaderboard. (The model we just built is actually a 1-layer neural network: more on this soon!)

What to Submit:

- **Part a:** Derivation of expression for $\widehat{W}$
- **Part b: Code** on Gradescope through coding submission
- **Part c:** Values of training and testing errors
- **Part d:** Display of 10 images whose labels are incorrectly predicted by the classifier. 1-2 sentences reasoning why.

Part A: Derivation

$$\widehat{W} = \min_{W \in R^{d \times k}} \sum_{i=1}^{n} \| W^T x_i - y_i \|_2^2 + \lambda \| W \|_F^2 \quad ¿ \sum_{j=1}^{k} \left[\sum_{i=1}^{n} \left(e_j^T W^T x_i - e_j^T y_i \right)^2 + \lambda \| W e_j \|_2^2 \right)$$

$$¿ \quad ¿ \sum_{j=1}^{k} \| X w_j - Y e_j \|_2^2 + \lambda \| w_j \|_2^2$$

where $ \mathbf{X} = \left[x_1 \dots x_n \right]^T \in \mathbb{R}^{n \times d} $ and $ \mathbf{Y} = \left[y_1 \dots y_n \right]^T \in \mathbb{R}^{n \times k} $. Show that

$$\widehat{W} = \left(X^T X + \lambda I \right)^{-1} X^T Y$$

set the gradient to zero:

$$0 = \frac{\partial}{\partial w_j} \widehat{W} = \frac{\partial}{\partial w_j} \left[\sum_{j=1}^{k} \| X w_j - y_j \|_2^2 + \lambda \| w_j \|_2^2 \right)$$

because the derivative of a squared norm $|Ax - b|_2^2 $ with respect to x is $ 2A^TAx - 2A^Tb $, and the derivative of $ |w_j|_2^2 $ with respect to $ w_j $ is $ 2w_j $. the equation becomes:

$$0 = 2 X^T X w_j - 2 X^T y_j + 2 \lambda w_j$$

factor out 2 and dividing both sides by 2, we get:

$$X^T X w_j - X^T y_j + \lambda w_j = 0$$

rearrange

$$\left(X^T X + \lambda I \right) w_j = X^T y_j$$

multiply both sides by the inverse of $ (X^T X + \lambda I) $:

$$w_j = \left(X^T X + \lambda I \right)^{-1} X^T y_j$$

if we stanck the w_j elements by rows we make $\widehat{W}$:

$$\widehat{W} = \begin{bmatrix} \left(X^T X + \lambda I \right)^{-1} \left(X^T y_1 \right) \\ \vdots \\ \left(X^T X + \lambda I \right)^{-1} \left(X^T y_k \right) \end{bmatrix} = \left(X^T X + \lambda I \right)^{-1} X^T Y$$

therefore:

$$\widehat{W} = \left(X^T X + \lambda I \right)^{-1} X^T Y$$

Part B: Code

```
#Brenton Mizell
#CSE 546
```

```python
import matplotlib.pyplot
import numpy as np

from utils import load_dataset, problem
#import for train "choochoo"
from scipy.linalg import solve
#for plots
import matplotlib.pyplot as plt



@problem.tag("hw1-A")
def train(x: np.ndarray, y: np.ndarray, _lambda: float) -> np.ndarray:
    """Train function for the Ridge Regression problem.
    Should use observations (`x`), targets (`y`) and regularization
parameter (`_lambda`)
    to train a weight matrix $$\\hat{W}$$.


    Args:
        x (np.ndarray): observations represented as `(n, d)` matrix.
            n is number of observations, d is number of features.
        y (np.ndarray): targets represented as `(n, k)` matrix.
            n is number of observations, k is number of classes.
        _lambda (float): parameter for ridge regularization.

    Raises:
        NotImplementedError: When problem is not attempted.

    Returns:
        np.ndarray: weight matrix of shape `(d, k)`
            which minimizes Regularized Squared Error on `x` and `y`
with hyperparameter `_lambda`.
    """
    #create a regularization matrix
    regulate = _lambda*np.eye(x.shape[1])
    #calculate and return the weight
    weight = solve((x.T @ x + regulate), (x.T @ y))
    return weight


@problem.tag("hw1-A")
def predict(x: np.ndarray, w: np.ndarray) -> np.ndarray:
    """Train function for the Ridge Regression problem.
    Should use observations (`x`), and weight matrix (`w`) to generate
predicated class for each observation in x.

    Args:
        x (np.ndarray): observations represented as `(n, d)` matrix.
```

```
            n is number of observations, d is number of features.
        w (np.ndarray): weights represented as `(d, k)` matrix.
            d is number of features, k is number of classes.

    Raises:
        NotImplementedError: When problem is not attempted.

    Returns:
        np.ndarray: predictions matrix of shape `(n,)` or `(n, 1)`.
    """
    #perform matrix multiplication
    #find max value along each row and return
    predictions = np.argmax(x @ w, axis=1)
    return predictions

@problem.tag("hw1-A")
def one_hot(y: np.ndarray, num_classes: int) -> np.ndarray:
    """One hot encode a vector `y`.
    One hot encoding takes an array of integers and coverts them into
binary format.
    Each number i is converted into a vector of zeros (of size
num_classes), with exception of i^th element which is 1.

    Args:
        y (np.ndarray): An array of integers [0, num_classes), of
shape (n,)
        num_classes (int): Number of classes in y.

    Returns:
        np.ndarray: Array of shape (n, num_classes).
        One-hot representation of y (see below for example).

    Example:
        ```python
        > one_hot([2, 3, 1, 0], 4)
        [
            [0, 0, 1, 0],
            [0, 0, 0, 1],
            [0, 1, 0, 0],
            [1, 0, 0, 0],
        ]
        ```
    """
    #create appropriately sized matrix of zeroes
    int_array = np.zeros((y.size, num_classes))
    #set the indicated columns per row equal to 1 and return
    int_array[np.arange(y.size), y] = 1
    return int_array
```

```
def main():

    (x_train, y_train), (x_test, y_test) = load_dataset("mnist")
    # Convert to one-hot
    y_train_one_hot = one_hot(y_train.reshape(-1), 10)

    _lambda = 1e-4

    w_hat = train(x_train, y_train_one_hot, _lambda)

    y_train_pred = predict(x_train, w_hat)
    y_test_pred = predict(x_test, w_hat)

    print("Ridge Regression Problem")
    print(
        f"\tTrain Error: {np.average(1 - np.equal(y_train_pred,
y_train)) * 100:.6g}%"
    )
    print(f"\tTest Error:  {np.average(1 - np.equal(y_test_pred,
y_test)) * 100:.6g}%")

    #show 10 samples of incorrectly labeled test data
    bad_predictions = np.where(y_test_pred!=y_test)[0]
    plt.figure(figsize=(20,4))
    for i, bad_prediction_index in enumerate(bad_predictions[:10]):
        plt.subplot(2,5,i+1)
        image = x_test[bad_prediction_index].reshape(28,28)
        plt.imshow(image,cmap='gray')
        plt.title(f"Prediction: {y_test_pred[bad_prediction_index]},
True: {y_test[bad_prediction_index]}")
        plt.axis('off')
        plt.tight_layout()
    plt.show()

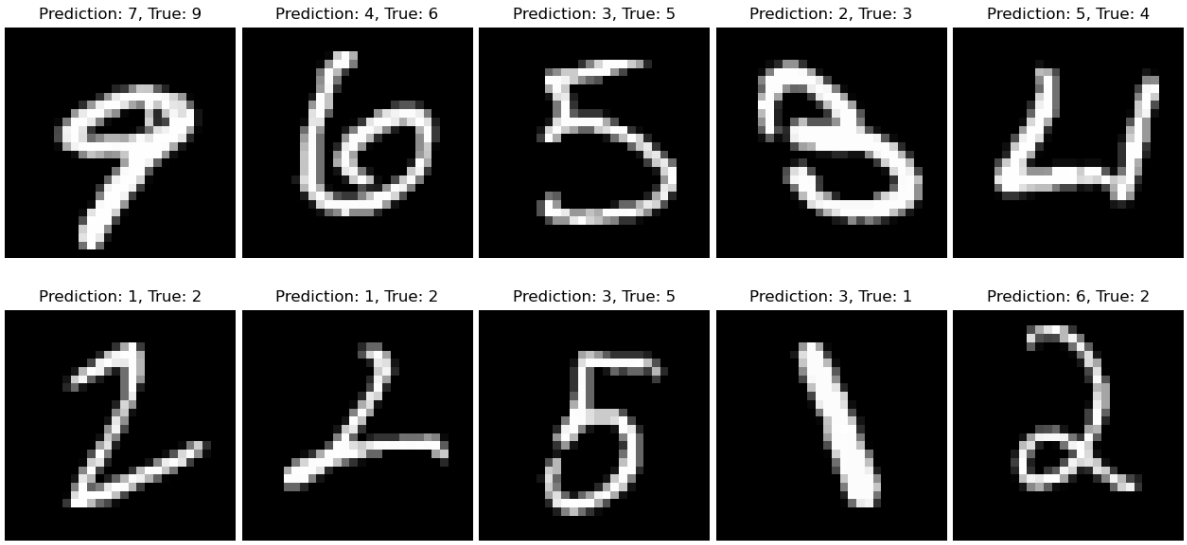

if __name__ == "__main__":
    main()
```

Part C: Values of training and testing errors

Train Error: 14.805%

Test Error: 14.66%

Part D: Display of 10 images whise labels are incorrectly predicted and why



My predicted reason on why the predicted images are incorrect is because when you are looking at the squares of the box, the white boxes forming the number do not appear where the average value of what is trained. They are slightly askew or they are oddly shaped making the white boxes, appear or dont appear in what is trained for a different value. I.e. these numbers above because of their odd shape more closely align to other trained values.

A6.

a. *[2 points]* **About how many hours did you spend on this homework? There is no right or wrong answer :)**