Homework #1

CSE 446/546: Machine Learning
Profs. Jamie Morgenstern and Ludwig Schmidt
Due: **Wednesday** October 19, 2022 11:59pm
A: 60 points, **B**: 30 points

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.

Solution:

a. Bias is how far off the correct parameter the estimator is on average. Variance is how spread out the estimator is. The bias-variance tradeoff is the idea that increasing the model complexity usually decreases bias but increases variance.
b. Increasing the model complexity usually decreases bias but increases variance.
c. False.
d. The validation set is used for hyperparameter tuning. The training set is used to train learned parameters (as opposed to hyperparameters), and the test set is only used to evaluate the error after training. The tuning procedure is to systematically vary each hyperparameter (e.g. learning rate in gradient descent), train on the training set, and set the hyperparameter to the value that gives the smallest error on the validation set.
e. False.

Maximum Likelihood Estimation (MLE)

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:

$$[2, 4, 6, 0, 1].$$

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 6 goals in their next game?

c. *[2 points]* Suppose the Reign 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 6 goals in their 7th game.

Solution:

a. We want to find λ that minimizes the log-likelihood function

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

Taking the derivative wrt λ and setting it to 0, we get

$$0 = \sum_{i=1}^{n} -1 + x_i/\lambda = -n + \frac{1}{\lambda}\sum_{i=1}^{n} x_i.$$

This gives

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

So our estimator is the average of the observed data.
b. My estimate is $(2+4+6+1)/5 = 13/5 = 2.6$. The estimated probability of scoring 6 goals is $e^{-2.6}\frac{2.6^6}{6!} \approx 0.0318671$.
c. My new estimate is $(2+4+6+1+8)/6 = 3.5$. The estimated probability of scoring 6 goals is $e^{-3.5}\frac{3.5^6}{6!} \approx 0.0770983$.

Overfitting

B1. Suppose we have N labeled samples $S = \{(x_i, y_i)\}_{i=1}^N$ drawn i.i.d. from an underlying distribution $\mathcal{D}$. Suppose we decide to break this set into a set S_{train} of size N_{train} and a set S_{test} of size N_{test} samples for our training and test set, so $N = N_{\text{train}} + N_{\text{test}}$, and $S = S_{\text{train}} \cup S_{\text{test}}$. Recall the definition of the true least squares error of f:

$$\epsilon(f) = \mathbb{E}_{(x,y)\sim\mathcal{D}}[(f(x) - y)^2],$$

where the subscript $(x, y) \sim \mathcal{D}$ makes clear that our input-output pairs are sampled according to $\mathcal{D}$. Our training and test losses are defined as:

$$\widehat{\epsilon}_{\text{train}}(f) = \frac{1}{N_{\text{train}}} \sum_{(x,y)\in S_{\text{train}}} (f(x) - y)^2$$

$$\widehat{\epsilon}_{\text{test}}(f) = \frac{1}{N_{\text{test}}} \sum_{(x,y)\in S_{\text{test}}} (f(x) - y)^2$$

We then train our algorithm using the training set to obtain $\widehat{f}$.

a. *[2 points]* (bias: the test error) For all fixed f (before we've seen any data) show that

$$\mathbb{E}_{\text{train}}[\widehat{\epsilon}_{\text{train}}(f)] = \mathbb{E}_{\text{test}}[\widehat{\epsilon}_{\text{test}}(f)] = \epsilon(f).$$

Use a similar line of reasoning to show that the test error is an unbiased estimate of our true error for $\widehat{f}$. Specifically, show that:

$$\mathbb{E}_{\text{test}}[\widehat{\epsilon}_{\text{test}}(\widehat{f})] = \epsilon(\widehat{f})$$

b. *[3 points]* (bias: the train/dev error) Is the above equation true (in general) with regards to the training loss? Specifically, does $\mathbb{E}_{\text{train}}[\widehat{\epsilon}_{\text{train}}(\widehat{f})]$ equal $\epsilon(\widehat{f})$? If so, why? If not, give a clear argument as to where your previous argument breaks down.

c. *[5 points]* Let $\mathcal{F} = (f_1, f_2, \dots)$ be a collection of functions and let $\widehat{f}_{\text{train}}$ minimize the training error such that $\widehat{\epsilon}_{\text{train}}(\widehat{f}_{\text{train}}) \leq \widehat{\epsilon}_{\text{train}}(f)$ for all $f \in \mathcal{F}$. Show that

$$\mathbb{E}_{\text{train}}[\widehat{\epsilon}_{\text{train}}(\widehat{f}_{\text{train}})] \leq \mathbb{E}_{\text{train,test}}[\widehat{\epsilon}_{\text{test}}(\widehat{f}_{\text{train}})].$$

(Hint: note that

$$\begin{aligned}
\mathbb{E}_{\text{train,test}}[\widehat{\epsilon}_{\text{test}}(\widehat{f}_{\text{train}})] &= \sum_{f\in\mathcal{F}} \mathbb{E}_{\text{train,test}}[\widehat{\epsilon}_{\text{test}}(f)\mathbf{1}\{\widehat{f}_{\text{train}} = f\}] \\
&= \sum_{f\in\mathcal{F}} \mathbb{E}_{\text{test}}[\widehat{\epsilon}_{\text{test}}(f)]\mathbb{E}_{\text{train}}[\mathbf{1}\{\widehat{f}_{\text{train}} = f\}] \\
&= \sum_{f\in\mathcal{F}} \mathbb{E}_{\text{test}}[\widehat{\epsilon}_{\text{test}}(f)]\mathbb{P}_{\text{train}}(\widehat{f}_{\text{train}} = f)
\end{aligned}$$

where the second equality follows from the independence between the train and test set.)

Solution:

a. By linearity of expectation,

$$\mathbb{E}[\widehat{\epsilon}_{\text{train}}(f)] = \mathbb{E}\left[\frac{1}{N_{\text{train}}} \sum_{(x,y)\in S_{\text{train}}} (f(x)-y)^2\right] = \frac{1}{N} \sum_{(x,y)\in S_{\text{train}}} \mathbb{E}(f(x)-y)^2$$
$$= \frac{N}{N}\mathbb{E}(f(x)-y)^2] = \mathbb{E}(f(x)-y)^2 = \epsilon(f).$$

The same exact proof works for the test set. Exchanging f for $\hat{f}$ in the proof above gives

$$\mathbb{E}_{\text{test}}[\widehat{\epsilon}_{\text{test}}(\widehat{f})] = \epsilon(\widehat{f}).$$

b. It is true. The above argument is simply linearity of expectation.
c. There is a lot of undefined notations here. For example, $\mathbb{E}_{\text{train,test}}$ and $\mathbb{P}_{\text{train}}$.

Bias-Variance tradeoff

B2. For $i = 1, \ldots, n$ let $x_i = i/n$ and $y_i = f(x_i) + \epsilon_i$ where $\epsilon_i \sim \mathcal{N}(0, \sigma^2)$ for some unknown f we wish to approximate at values $\{x_i\}_{i=1}^n$. We will approximate f with a step function estimator. For some $m \leq n$ such that n/m is an integer define the estimator

$$\widehat{f}_m(x) = \sum_{j=1}^{n/m} c_j \mathbf{1}\{x \in \left(\frac{(j-1)m}{n}, \frac{jm}{n}\right]\} \quad \text{where} \quad c_j = \frac{1}{m} \sum_{i=(j-1)m+1}^{jm} y_i.$$

Note that $x \in \left(\frac{(j-1)m}{n}, \frac{jm}{n}\right]$ means x is in the open-closed interval $\left(\frac{(j-1)m}{n}, \frac{jm}{n}\right]$.

Note that this estimator just partitions $\{1, \ldots, n\}$ into intervals $\{1, \ldots, m\}, \{m+1, \ldots, 2m\}, \ldots, \{n-m+1, \ldots, n\}$ and predicts the average of the observations within each interval.
By the bias-variance decomposition at some x_i we have

$$\mathbb{E}\left[(\widehat{f}_m(x_i) - f(x_i))^2\right] = \underbrace{(\mathbb{E}[\widehat{f}_m(x_i)] - f(x_i))^2}_{\text{Bias}^2(x_i)} + \underbrace{\mathbb{E}\left[(\widehat{f}_m(x_i) - \mathbb{E}[\widehat{f}_m(x_i)])^2\right]}_{\text{Variance}(x_i)}$$

a. *[5 points]* Intuitively, how do you expect the bias and variance to behave for small values of m? What about large values of m?

b. *[5 points]* If we define $\bar{f}^{(j)} = \frac{1}{m}\sum_{i=(j-1)m+1}^{jm} f(x_i)$ and the *average bias-squared* as

$$\frac{1}{n}\sum_{i=1}^{n}(\mathbb{E}[\widehat{f}_m(x_i)] - f(x_i))^2 \ ,$$

show that

$$\frac{1}{n}\sum_{i=1}^{n}(\mathbb{E}[\widehat{f}_m(x_i)] - f(x_i))^2 = \frac{1}{n}\sum_{j=1}^{n/m}\sum_{i=(j-1)m+1}^{jm}(\bar{f}^{(j)} - f(x_i))^2$$

c. *[5 points]* If we define the *average variance* as $\mathbb{E}\left[\frac{1}{n}\sum_{i=1}^{n}(\widehat{f}_m(x_i) - \mathbb{E}[\widehat{f}_m(x_i)])^2\right]$, show (both equalities)

$$\mathbb{E}\left[\frac{1}{n}\sum_{i=1}^{n}(\widehat{f}_m(x_i) - \mathbb{E}[\widehat{f}_m(x_i)])^2\right] = \frac{1}{n}\sum_{j=1}^{n/m} m\mathbb{E}[(c_j - \bar{f}^{(j)})^2] = \frac{\sigma^2}{m}$$

d. *[5 points]* By the Mean-Value theorem we have that

$$\min_{i=(j-1)m+1,\ldots,jm} f(x_i) \leq \bar{f}^{(j)} \leq \max_{i=(j-1)m+1,\ldots,jm} f(x_i)$$

Suppose f is L-Lipschitz[a] so that $|f(x_i) - f(x_j)| \leq \frac{L}{n}|i-j|$ for all $i, j \in \{1, \ldots, n\}$ for some $L > 0$.

Show that the average bias-squared is $O(\frac{L^2m^2}{n^2})$. Using the expression for average variance above, the total error behaves like $O(\frac{L^2m^2}{n^2} + \frac{\sigma^2}{m})$. Minimize this expression with respect to m.

Does this value of m, and the total error when you plug this value of m back in, behave in an intuitive way with respect to n, L, σ^2? That is, how does m scale with each of these parameters? It turns out that this simple estimator (with the optimized choice of m) obtains the best achievable error rate up to a universal constant in this setup for this class of L-Lipschitz functions (see Tsybakov's *Introduction to Nonparametric Estimation* for details).[b]

Solution:

a. The complexity of the model increases with m. So I would expect the bias to be high and variance to be low for small m, and the bias to be low with high variance for large m.

[a] A function is L-Lipschitz if there exists $L \geq 0$ such that $||f(x_i) - f(x_j)|| \leq L||x_i - x_j||$, for all x_i, x_j

[b] This setup of each x_i deterministically placed at i/n is a good approximation for the more natural setting where each x_i is drawn uniformly at random from $[0, 1]$. In fact, one can redo this problem and obtain nearly identical conclusions, but the calculations are messier.

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, I would recommend the closed-form solution since the data sets are small; for this reason, we've included an example closed-form implementation of 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 actually 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.

   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 $x^8 = 2.56 \times 10^{10}$ – 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).

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

Solution:

a.

```
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
        # constants for normalization
        self.mean = 0
        self.std = 1

    @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.

        """
        n, _ = X.shape
        new_X = np.ones([n, degree])
        for i in range(1, degree+1):
            new_X[:,i-1] = (X**i)[:,0]

        return new_X

    @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 need to apply polynomial expansion and scaling at first.
        """
        n = len(X)
        poly_X = self.polyfeatures(X, self.degree)
        self.std = np.std(poly_X, axis = 0)
        self.mean = np.mean(poly_X, axis = 0)
        X_ = (poly_X - self.mean)/(self.std)
        X_ = np.c_[np.ones([n, 1]), X_]
        # construct reg matrix
        reg_matrix = self.reg_lambda * np.eye(self.degree+1)
        reg_matrix[0, 0] = 0

        # analytical solution (X'X + regMatrix)^-1 X' y
        self.weight = np.linalg.solve(X_.T @ X_ + reg_matrix, X_.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.
        """
        n = len(X)

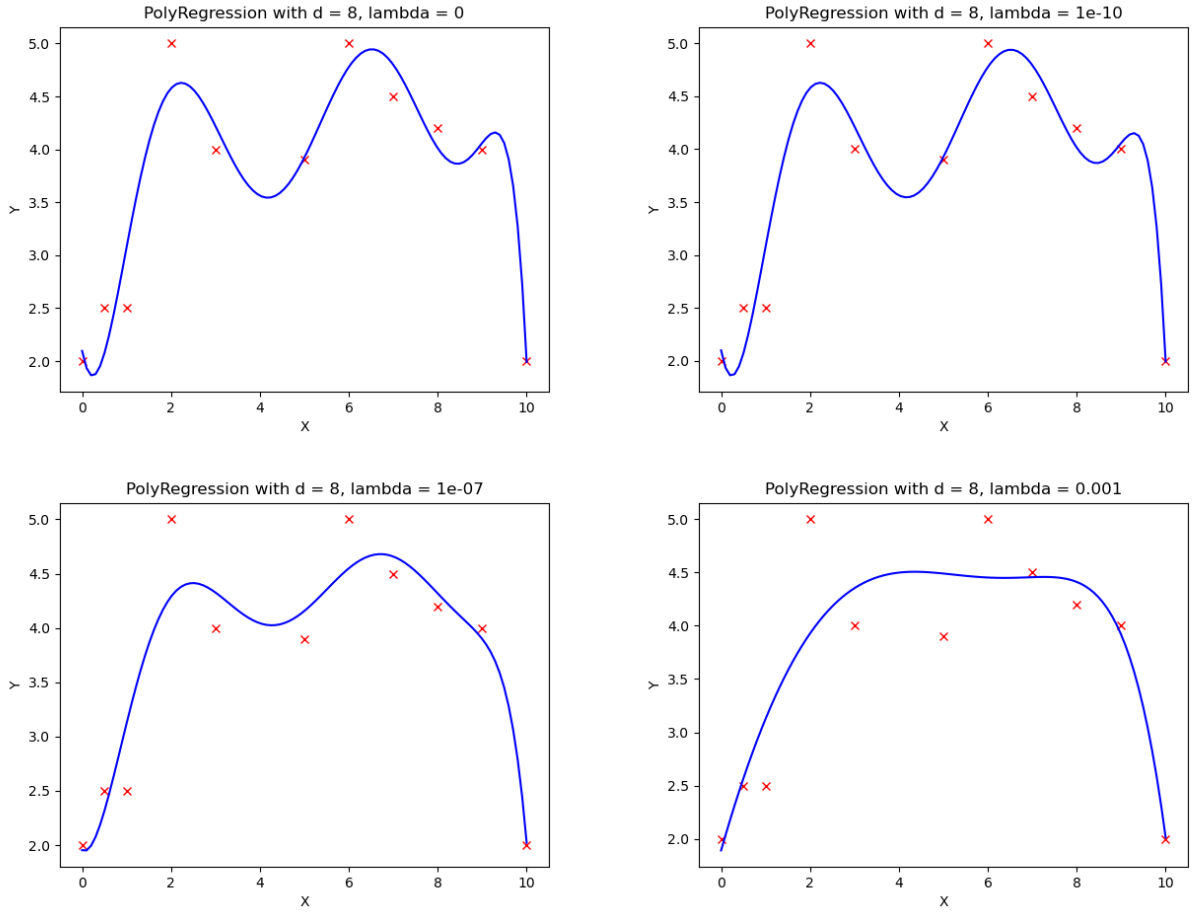
        poly_X = self.polyfeatures(X, self.degree)

        X_ = (poly_X - self.mean)/(self.std)
        X_ = np.c_[np.ones([n, 1]), X_]

        return X_.dot(self.weight)
```

b. The function gets smoother with more regularization. It looks like $\lambda = 0.001$ is too much regularization while $\lambda = 10^{-7}$ looks good.

Figure 1: Increasing λ



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

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

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.
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.

Solution:

a.

```
@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 np.linalg.norm(a-b)**2/len(a)


@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 Xtra
            2. errorTest -- errorTest[i] is the testing mean squared error using model trained by Xtrain[

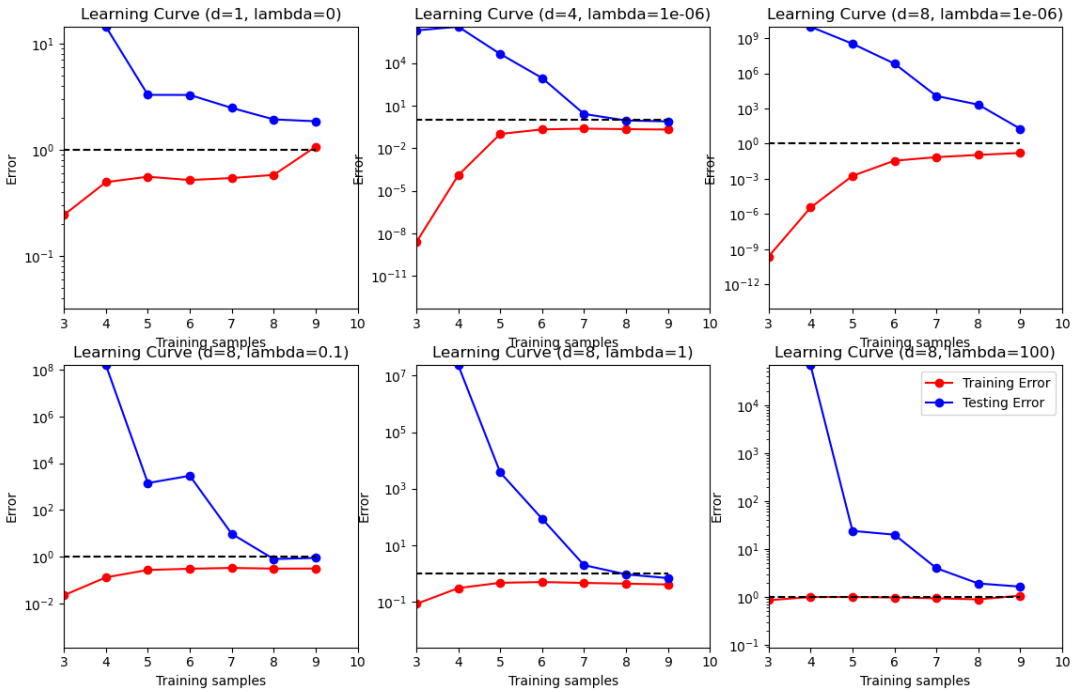
    Note:
        - For errorTrain[i] only calculate error on Xtrain[0:(i+1)], since this is the data used for trai
            THIS DOES NOT APPLY TO errorTest.
        - errorTrain[0:1] and errorTest[0:1] won't actually matter, since we start displaying the learnin
    """
    n = len(Xtrain)

    errorTrain = np.zeros(n)
    errorTest = np.zeros(n)
    # Fill in errorTrain and errorTest arrays
    for i in range(2,n):
        model = PolynomialRegression(degree=degree, reg_lambda=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
```

b.

Figure 2: Learning curves



Ridge Regression on MNIST

Relevant Files[2]:

- **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`).

Each example has features $x_i \in \mathbb{R}^d$ (with $d = 28 * 28 = 784$) and label $z_j \in \{0, \ldots, 9\}$. 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 **??** 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

$$\sum_{i=1}^{n} \|W^T x_i - y_i\|_2^2 + \lambda \|W\|_F^2 = \sum_{j=1}^{k} \left[\sum_{i=1}^{n} (e_j^T W^T x_i - e_j^T y_i)^2 + \lambda \|W e_j\|^2 \right] \tag{3}$$

$$= \sum_{j=1}^{k} \left[\sum_{i=1}^{n} (w_j^T x_i - e_j^T y_i)^2 + \lambda \|w_j\|^2 \right] \tag{4}$$

$$= \sum_{j=1}^{k} \left[\|X w_j - Y e_j\|^2 + \lambda \|w_j\|^2 \right] \tag{5}$$

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

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

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

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 $Y \in \{0, ..., k-1\}^n$, 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 $\arg\max_{j=0,...,9} e_j^T W^T x_i'$ 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?

Once you finish this problem question, you should have a very powerful classifier for handwritten digits! 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!)

Solution:

a. Using equation (5), it suffices to minimize each term in the sum separately, i.e.

$$\hat{w}_j = \text{argmin}_{w \in \mathbb{R}^{d \times 1}} \|Xw - Ye_j\|^2 + \lambda \|w\|^2.$$

Since $Ye_j \in \mathbb{R}^n$, we know from lecture that the minimizer of the above is given by

$$\hat{w}_j = (X^T X + \lambda I)^{-1} X^T (Ye_j).$$

By associativity of matrix multiplication,

$$\hat{w}_j = \left((X^T X + \lambda I)^{-1} X^T Y\right) e_j = \left((X^T X + \lambda I)^{-1} X^T Y\right)_j.$$

So we get the desired expression for $\hat{W}$:

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

b.

```
import numpy as np

from utils import load_dataset, problem


@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 ff\\hat{W}ff.


    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`.
    """
    n, d = x.shape
    reg_matrix = _lambda * np.eye(d)

    # analytical solution (X'X + regMatrix)^-1 X' y
    return np.linalg.solve(x.T @ x + reg_matrix, x.T @ y)


@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 predicted class for each observati

    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)`.
    """
    return np.argmax(x.dot(w), axis = 1)


@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 eleme

    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],
        ]
        ```
    """
    result = np.zeros([len(y), num_classes])
    for (i, yi) in enumerate(y):
        result[i, yi] = 1.0 # assumes that elements of y are in set {0,...,num_classes-1}
    return result

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}%")

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

c. Train Error: 14.805
Test Error: 14.66

# Administrative

A6.

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

Solution: I spent about 10 hours.
```