Homework #1

CSE 546: Machine Learning
Raman SV

List of collaborators and problem(s) collaborated on: No collaborators this time

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?

(a) Answer:
a. Bias can be defined as the ability to “accurately” capture the true underlying relationship between the input features (predictor) and output response (target). Bias is the holdout in creating accurate models that truly represent the underlying data – the training set.
b. Variance can be defined as the ability to “generalize” the above created model to new and unseen data. Variance is the difference in fit or performance between the model we developed based on the training set and the real or unseen data (test data or in cases of deployment, actual user data) the model encounters.
c. Based on the above 2 statements, bias-variance tradeoff can be thought of as the tradeoff between creating a model based on how it performs on the training data as compared to its efficacy on real data (post training). It is about finding the equilibrium in a model between a great fit on the training data and its performance on real world or test data.

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

(a) Answer:
a. As model complexity increases, bias decreases monotonically and variance increases
b. Conversely, as model complexity decreases, bias increases monotonically and variance decreases

This occurs because as the completxity increases, the model fits the training data more accurately and hence bias reduces, while its generalizing ability on new data sets reduces, which is why variance increases.

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.

(a) Answer:
a. False. As we collect more data from the same distribution, the predictor models the training data better and can more accurately fit the patterns in the distribution. Thus, this reduces the variance of the predictor.

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.

(a) Answer:
a. Validation set should be used for hyperparameter training as we never train the data on the test set. Further, the model itself is trained on the train set and hence we can use the validation set to pick a hyperparameter based on the data.
b. One procedure for hyperparameter tuning is via the k-fold cross validation strategy. We randomly divide the training data into k equal parts and learn the classifier for k-1 training sets and examine its performance on the 1 validation set. The k-fold cross validation error would be the average over data splits. We can repeat this process till we achieve optimal tradeoff for the hyperparameter.

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.

(a) Answer:
False. Typically, a function is modelled/set up to fit the training set. Thus, the function is likely to perform better on the training set. This implies that the training error of a function on the training set would be an underestimate of the true error instead.

What to Submit:

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

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

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

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.)

   (a) Answer:
   We are given that $x_1, \ldots, x_5$ are iid. The PMF is given as

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

   Therefore, the likelihood function, $L_n(\lambda) = \prod_{i=1}^{n} f(X_i; \lambda)$
   $= \prod_{i=1}^{n} e^{-\lambda}\frac{\lambda^{x_i}}{x_i!}$
   The log-likelihood function is -
   $l_n(\lambda) = log(L_n(\lambda)) = \sum_{i=1}^{n} log(e^{-\lambda}\frac{\lambda^{x_i}}{x_i!})$

   $= \sum_{i=1}^{n} log(e^{-\lambda}) - log(x_i!) + log(\lambda^{x_i})$

   $= -n\lambda - \sum_{i=1}^{n} log(x_i!) + log(\lambda)\sum_{i=1}^{n} x_i) \ldots (1)$

   The MLE for λ is $\widehat{\lambda}_{MLE} = argmax_{\lambda} L_n(\lambda)$
   To calculate the MLE, we take the first derivative of the log likelihood function with respect to λ and equate it to 0

   Taking the derivative wrt λ for $\ldots$ (1), we have,
   $-n + \frac{1}{\widehat{\lambda}_{MLE}}\sum_{i=1}^{n} x_i = 0$

   $\therefore \widehat{\lambda}_{MLE} = \frac{1}{n}\sum_{i=1}^{n} x_i$

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?

   (a) Answer:
   Based on the $\widehat{\lambda}_{MLE}$ calculated in the above section, the numerical estimate after the first 5 games is $\frac{(2+4+6+0+1)}{5} = 2.6$

Based on the above value of λ, the probability that Reign score exactly 6 goals in the next game is **0.032**

$\text{Poi}(6|2.6) = e^{-2.6}\frac{2.6^6}{6!} = 0.032$

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 exactly 6 goals in their 7th game.

(a) Answer:
Based on the $\widehat{\lambda}_{MLE}$ calculated in the section a, the updated numerical estimate after 6 games is $\frac{(2+4+6+0+1+8)}{6} = 3.5$

Based on the above value of λ, the probability that Reign score exactly 6 goals in the 7th game is **0.077**

$\text{Poi}(6|3.5) = e^{-3.5}\frac{3.5^6}{6!} = 0.077$

What to Submit:

- **Part a:** An expression for the MLE of λ after n games and relevant derivation
- **Parts b-c:** A numerical estimate for λ and the probability that the Reign score 6 next game.

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.)

What to Submit:

- **Part a:** Proof
- **Part b:** Brief Explanation (3-5 sentences)
- **Part c:** Proof

Bias-Variance tradeoff

B2. For $i = 1, \dots, n$ 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 $\{1, \dots, n\}$ into intervals $\{1, \dots, m\}, \{m+1, \dots, 2m\}, \dots, \{n-m+1, \dots, n\}$ and predicts the average of the observations within each interval (see Figure 1).

`../img/step_function.png`

Figure 1: Step function estimator with $n = 256$, $m = 16$, and $\sigma^2 = 1$.

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

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

Suppose f is L-Lipschitz[a] so that $|f(x_i) - f(x_j)| \le \frac{L}{n}|i - j|$ for all $i, j \in \{1, \dots, n\}$ 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]

What to Submit:

- **Part a:** 1-2 sentences
- **Part b:** Proof
- **Part c:** Proof
- **Part d:** Derivation of minimal error with respect to m. 1-2 sentences about scaling of m with parameters.

[a] A function is L-Lipschitz if there exists $L \ge 0$ such that $||f(x_i) - f(x_j)|| \le L||x_i - x_j||$, 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]*

(a) Answer:

```
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
    self.fit_mean: np.ndarray = None  # type: ignore
    self.fit_std: np.ndarray = None  # type: ignore
    # You can add additional fields
    # self.theta = None
    # raise NotImplementedError("Your Code Goes Here")

@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.
```

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

```
        """
        n, d1 = X.shape
        if degree < 1:
            pfX_ = np.zeros((1, 1))
        else:
            if d1 != 1:
                pfX_ = np.zeros((1, 1))
            else:
                pfX_ = np.zeros((n, degree))
                for i in range(n):
                    for j in range(degree):
                        pfX_[i, j] = X[i] ** (j + 1)
        return pfX_

    @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.
        """
        # 1. applying polynomial expansion to the input data
        # print(f'shape of input data is {X.shape}')
        X_ = self.polyfeatures(X, self.degree)

        # 2. Standardizing the expanded matrix
        X_std = (X_ - np.mean(X_, axis=0)) / (np.std(X_, axis=0))
        self.fit_mean = np.mean(X_, axis=0)
        self.fit_std = np.std(X_, axis=0)
        # 3. Adding the bias term
        n1, d1 = X_std.shape
        '''
        newdata = np.zeros((n1, d1 + 1))
        newdata[:, 0] = 1
        for i in range(n1):
            for j in range(d1):
                newdata[i, j + 1] = X_std[i, j]
        '''

        newdata = np.c_[np.ones([n1, 1]), X_std]

        # 4. Solving for the coefficients
        reg_matrix = self.reg_lambda * np.eye(d1 + 1)
        reg_matrix[0, 0] = 0
        # print(f'shape of newdata is is {newdata.shape}')
        np.savetxt('input.csv', newdata, fmt="%f", delimiter=",")

        # analytical solution (X'X + regMatrix)^-1 X' y
```

```
            self.weight = np.linalg.solve(newdata.T @ newdata + reg_matrix,
            newdata.T @ y)
            # print(self.theta.shape)
            # print(f'theta values are {self.theta}')

            # raise NotImplementedError("Your Code Goes Here")

        @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.
            """
            # print(f'shape of test data is {X.shape}')
            # n = len(X)
            X_ = self.polyfeatures(X, self.degree)
            X_std = (X_ - self.fit_mean) / (self.fit_std)
            # X_std = (X - np.mean(X, axis=0)) / (np.std(X, axis=0))

            # print(f'the standardized input is \n {X_std}')

            # X_std.tofile('output.csv', sep=',', format='%10.5f')
            #np.savetxt('output.csv', X_std, fmt="%f", delimiter=",")


            # add 1s column
            # X_ = np.c_[np.ones([n, 1]), X_std]
            n1, d1 = X_std.shape
            '''
            newdata = np.zeros((n1, d1 + 1))
            newdata[:, 0] = 1

            for i in range(n1):
                for j in range(d1):
                    newdata[i, j + 1] = X_std[i, j]
            '''
            newdata = np.c_[np.ones([n1, 1]), X_std]

            # predict
            return newdata.dot(self.weight)
            # raise NotImplementedError("Your Code Goes Here")


    @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.
    """
    if a.shape != b.shape:
        return -1
    else:
        return np.square(np.subtract(a, b)).mean()
```

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).

   (a) Answer:
   We notice that as the regularization increases the curve becomes smoother and hence fits lesser data points in the training set. At 0 regularization, the function fits most of the data points but might not be very general. However, as we increase the regularization parameter value, we notice that the model now fits much lesser data but has a more regular shape. This can be seen below in the 4 plots for increasing value of the regularization parameter -

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.

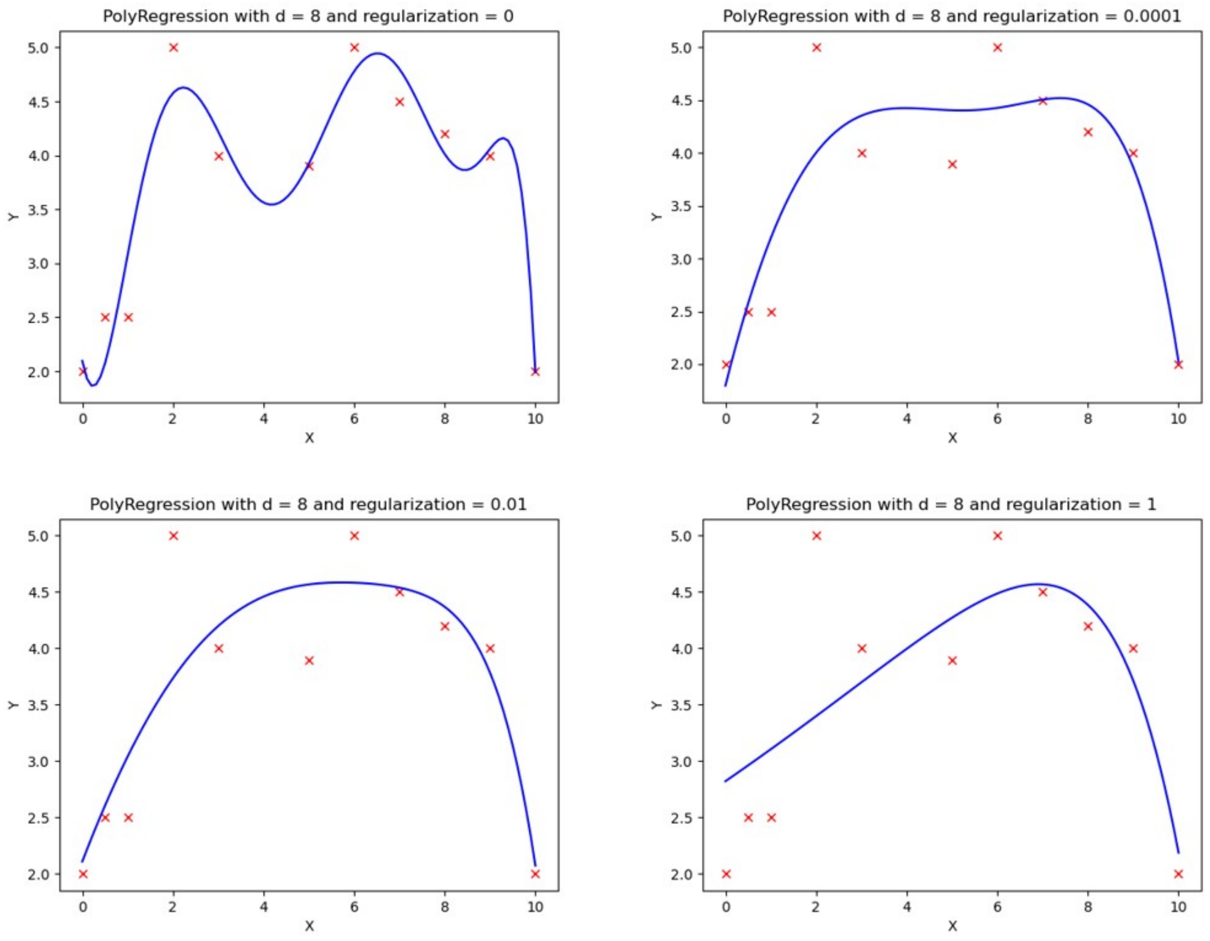

Figure 2: Observation of increasing Regularization parameter

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.

a. Answer: The plot from my code is below -

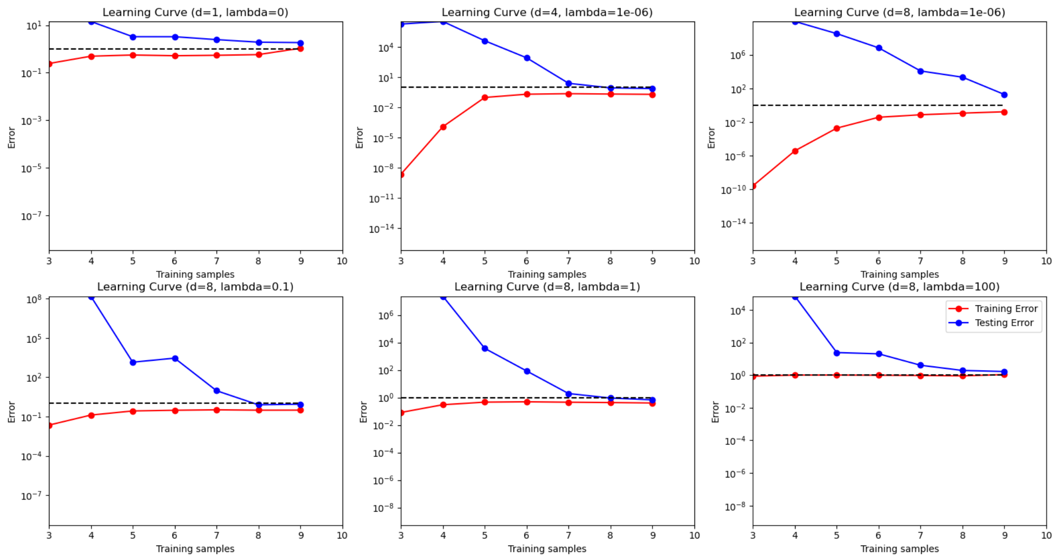

Figure 3: Observation of increasing Regularization parameter

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

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.

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 $\arg\max_{j=0,\dots,9} e_{j+1}^T \widehat{W}^T x_i$. Note that if $W = \begin{bmatrix} w_1 & \dots & w_k \end{bmatrix}$ then

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

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

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

(a) Answer :
we are given that the linear classifier to minimize the regularized least squares here is -

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

Expanding this term and accounting for e_j with $j \in 0\dots 9$ being a vector that is all zeros except for a 1 in the j^{th} position, we have -

$$\sum_{i=1}^{n} \|W^T x_i - y_i\|_2^2 + \lambda\|W\|_F^2 = \sum_{j=1}^{k}\left[\|Xw_j - Ye_j\|^2 + \lambda\|w_j\|^2\right]$$

We can further expand the squares inside as below -

$$= \sum_{j=1}^{k} \left[(Xw_j - Ye_j)^T(Xw_j - Ye_j) + \lambda w_j^T I w_j\right]$$

(we include the Identity matrix as this is the ridge regression setup)

$$= \sum_{j=1}^{k} \left[(Xw_j)^T Xw_j - (Xw_j)^T(Ye_j) - (Ye_j)^T(Xw_j) + (Ye_j)^T(Ye_j) + \lambda w_j^T I w_j\right]$$

$$= \sum_{j=1}^{k} \left[w_j^T X^T X w_j - 2w_j^T X^T(Ye_j) + (Ye_j)^T(Ye_j) + \lambda w_j^T I w_j\right]$$

$$= \sum_{j=1}^{k} \left[w_j^T (X^T X + \lambda I) w_j - 2w_j^T X^T(Ye_j) + (Ye_j)^T(Ye_j)\right]$$

To minimize the regularized least squares above, we need to minimize w_j as in the above equation, w_j is the value that minimizes W.
Thus, we take the gradient (first order derivative) with respect to w_j and equate it to 0

$$0 = \sum_{j=1}^{k} \left[\nabla_{w_j}(w_j^T (X^T X + \lambda I) w_j - 2w_j^T X^T(Ye_j) + (Ye_j)^T(Ye_j))\right]$$

$$0 = \sum_{j=1}^{k} \left[\nabla_{w_j}(w_j^T (X^T X + \lambda I) w_j) - \nabla_{w_j}(2w_j^T X^T(Ye_j)) + 0\right]$$

$$0 = \sum_{j=1}^{k} \left[2(X^T X + \lambda I) w_j - 2X^T(Ye_j)\right]$$

Applying the summation above and solving the equation, we have -

$$2(X^T X + \lambda I)\widehat{W} = 2X^T Y$$

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

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 $Y \in \{0, ..., k-1\}^n$, 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.

- Answer: Below is my code for the train and one hot functions

```
                    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`.
    """
    n, d = x.shape
    lambda_matrix = lambda * np.identity(d, dtype="int")
    weight = np.linalg.solve(x.T @ x + lambda_matrix, x.T @ y)
    return weight

    #raise NotImplementedError("Your Code Goes Here")


@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)`.
    """
    raise NotImplementedError("Your Code Goes Here")


@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],
    ]
    ```
"""
n = len(y)
temp_one = np.zeros((n, num_classes))
for i in range(n):
    k = y[i]
    temp_one[i,k] = 1
return temp_one
#raise NotImplementedError("Your Code Goes Here")
~~~

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.

Administrative

A6.

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

(a) I spent around 30 hours on this homework -
A1 - 0.5 hours
A2 - 1.5 hours
A3 - 10 hours
A4 - 4 hours
A5 - 7 hours
B1 - 1 hours
B2 - 2 hours
Latex type setting and general reading related to the topic - 5 hours