Homework #1

CSE 446/546: Machine Learning
Prof. Kevin Jamieson and Prof. Simon S. Du
Due: **Wednesday** April 19, 2023 11:59pm
A: 62 points, **B**: 30 points
Solutions by: EJ Rainville
Collaborators: Jake Davis

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?

Solution. **Bias** is the difference between the average predicted value and the true value and represents the accuracy of your model to predict the correct value. While **variance** represents the precision of the model where a low variance model will predict close to the same value for input data close to each other while high variance will have a larger spread for closely spaced input values. The bias-variance trade-off is the relationship between model complexity and error. In an ideal world, you could have a low bias and low variance model but generally you must sacrifice one for the other due to added complexity of the model.

---

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

Solution. Simple models tend to have higher bias but low variance while highly complex models have much lower bias but much higher variance. Due to this as you increase the complexity of your model, you are reducing the bias(increasing accuracy) of the model but increasing the variance (losing precision).

---

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.

Solution. **False.** The variance of the algorithm is due to the complexity of the model which means how many features are included rather than the number of data points collected. Collecting more data will help to have a more robust training/test dataset but will not affect the variance of the model.

---

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.

Solution. You should use the **validation** dataset to tune the hyperparameters of the model. This could be done where you train your model on the test dataset for a range of hyperparameters and then compute the error of the model when predicting the validation dataset. You can now plot the error in the validation dataset as a function of the hyperparameter and choose the hyperparameter that gives the lowest error on the validation dataset.

---

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. **False.** The training error is optimistically biased since we are intentionally minimizing the training error. Therefore, the true error is generally larger than the training error. If the test dataset is not used at all in training then the test error can act as an unbiased estimator of the true error.

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.)

Solution. To find the maximum-likelihood estimate of the parameter λ we will start by with the given probability distribution and we can define the MLE as the following.

$$\hat{\lambda} = argmax_\lambda P(X|\lambda) \tag{1}$$

since the scores of each game are independent and identically distributed we can write the total probability as the product of the probability of all events x_i as the following.

$$\hat{\lambda} = argmax_\lambda \prod_{i=1}^{n} p(x_i|\lambda) \tag{2}$$

Then computing the logarithm of the likelihood we can simplify to,

$$\hat{\lambda} = argmax_\lambda \sum_{i=1}^{n} log(p(x_i|\lambda)) \tag{3}$$

Now substituting the probability model of the Poisson distribution we get the following.

$$\hat{\lambda} = argmax_\lambda \sum_{i=1}^{n} log(e^{-\lambda}\frac{\lambda_i^x}{x_i!}) \tag{4}$$

Now taking the gradient with respect to λ and setting equal to zero we find the following.

$$0 = \frac{\partial}{\partial\lambda}[\sum_{i=1}^{n} log(e^{-\lambda}\frac{\lambda_i^x}{x_i!})] \tag{5}$$

$$= \sum_{i=1}^{n} \frac{\partial}{\partial\lambda}[log(e^{-\lambda}) + log(\frac{\lambda_i^x}{x_i!})] \tag{6}$$

$$= \sum_{i=1}^{n} \frac{\partial}{\partial\lambda}[-\lambda + log(\frac{\lambda_i^x}{x_i!})] \tag{7}$$

$$= \sum_{i=1}^{n} [\frac{\partial}{\partial\lambda}(-\lambda) + \frac{\partial}{\partial\lambda} log(\frac{\lambda_i^x}{x_i!})] \tag{8}$$

$$= \sum_{i=1}^{n} [-1 + \frac{x_i}{\lambda}] \tag{9}$$

$$= -n + \frac{1}{\lambda}\sum_{i=1}^{n} x_i \tag{10}$$

$$\tag{11}$$

Rearranging and solving for lambda, we find that the maximum likelihood estimator is,

$$\hat{\lambda} = \frac{1}{n}\sum_{i=1}^{n} x_i \tag{12}$$

---

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?

Solution. After the first five games we can compute the parameter $\hat{\lambda}$ as the following,

$$\lambda_5 = \frac{1}{5}(2 + 4 + 6 + 0 + 1) \tag{13}$$

$$= 2.6 \tag{14}$$

Now computing the probability that the Reign score exactly 6 in the next is the following,

$$P(6|2.6) = e^{-2.6}\frac{2.6^6}{6!} \tag{15}$$

$$= 0.031 \approx 3.1\% \tag{16}$$

Therefore, under this model the Reign have a 3.1% chance of scoring exactly 6 goals given their previous scores.

---

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.

Solution. After the six games we can compute the parameter $\hat{\lambda}$ as the following,

$$\lambda_6 = \frac{1}{5}(2 + 4 + 6 + 0 + 1 + 8) \tag{17}$$

$$= 3.5 \tag{18}$$

Now computing the probability that the Reign score exactly 6 in the next is the following,

$$P(6|3.5) = e^{-3.5}\frac{3.5^7}{7!} \tag{19}$$

$$= 0.038 \approx 3.8\% \tag{20}$$

Therefore, under this model the Reign have a 3.8% chance of scoring exactly 6 goals given their previous scores.

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 $\hat{f}$. Specifically, show that:

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

Solution. Lets first start with the furthest LHS of the first equation and substitute the definition of the training error as follows,

$$\mathbb{E}[\hat{\epsilon_{train}}(f)] = \mathbb{E}\left[\frac{1}{N_{train}} \sum_{(x,y)\in S_{train}} (f(x) - y)^2\right] \tag{21}$$

Through the linearity of expectation we can rearrange this be the following,

$$\mathbb{E}[\hat{\epsilon_{train}}(f)] = \frac{1}{N_{train}} \sum_{(x,y)\in S_{train}} \mathbb{E}\left[(f(x) - y)^2\right] \tag{22}$$

Then substituting the definition of the true error we find the following,

$$\mathbb{E}[\hat{\epsilon_{train}}(f)] = \frac{1}{N_{train}} \sum_{(x,y)\in S_{train}} \epsilon(f) \tag{23}$$

$$= \frac{N_{train}}{N_{train}} \epsilon(f) \tag{24}$$

$$= \epsilon(f) \tag{25}$$

Using this exact same procedure for the test dataset we find the same result and therefore we see,

$$\mathbb{E}_{\text{train}}[\widehat{\epsilon}_{\text{train}}(f)] = \mathbb{E}_{\text{test}}[\widehat{\epsilon}_{\text{test}}(f)] = \epsilon(f) \tag{26}$$

Now looking at the second part of this question, lets start on the left hand side as well and again substituting the definition of the test error we find the following,

$$\mathbb{E}\Big[\hat{\epsilon_{test}}(\hat{f})\Big] = \mathbb{E}\left[\frac{1}{N_{test}} \sum_{(x,y)\in S_{test}} (\hat{f}(x) - y)^2\right] \tag{27}$$

$$= \frac{1}{N_{test}} \sum_{(x,y)\in S_{test}} \mathbb{E}\Big[(\hat{f}(x) - y)^2\Big] \tag{28}$$

$$= \frac{N_{test}}{N_{test}} \epsilon(\hat{f}) \tag{29}$$

$$= \epsilon(\hat{f}) \tag{30}$$

Which is what we expect.

---

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.

Solution. This is not true with regards to the training loss. The training error in general should be less than the true error or the test error. This is since the function, f, is being minimized to train the model based on the data in the training set than the error is being intentionally minimized for this dataset which might not generalize for the test and true error.

---

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. Lets first start with the RHS and using the hint above we know the following,

$$\mathbb{E}_{\text{train,test}}[\widehat{\epsilon}_{\text{test}}(\widehat{f}_{\text{train}})] = \sum_{f\in\mathcal{F}} \mathbb{E}_{\text{test}}[\widehat{\epsilon}_{\text{test}}(f)]\mathbb{P}_{\text{train}}(\widehat{f}_{\text{train}} = f) \tag{31}$$

From part A we also found the following,

$$\mathbb{E}_{\text{test}}[\widehat{\epsilon}_{\text{test}}(f)] = \mathbb{E}_{\text{train}}[\widehat{\epsilon}_{\text{train}}(f)] \tag{32}$$

Now substituting this into the RHS from above we get the following,

$$\mathbb{E}_{\text{train,test}}[\widehat{\epsilon}_{\text{test}}(\widehat{f}_{\text{train}})] = \sum_{f\in\mathcal{F}} \mathbb{E}_{\text{train}}[\widehat{\epsilon}_{\text{train}}(f)]\mathbb{P}_{\text{train}}(\widehat{f}_{\text{train}} = f) \tag{33}$$

From the fact given, $\widehat{\epsilon}_{\text{train}}(\widehat{f}_{\text{train}}) \leq \widehat{\epsilon}_{\text{train}}(f)$, we can apply the expectation to both sides resulting in the following,

$$\mathbb{E}_{\text{train}}[\widehat{\epsilon}_{\text{train}}(\widehat{f}_{\text{train}})] \leq \mathbb{E}_{\text{train}}[\widehat{\epsilon}_{\text{train}}(f)] \tag{34}$$

Therefore, the following must be true.

$$\sum_{f \in \mathcal{F}} \mathbb{E}_{\text{train}}[\widehat{\epsilon}_{\text{train}}(\widehat{f}_{\text{train}})]\mathbb{P}_{\text{train}}(\widehat{f}_{\text{train}} = f) \leq \sum_{f \in \mathcal{F}} \mathbb{E}_{\text{train}}[\widehat{\epsilon}_{\text{train}}(f)]\mathbb{P}_{\text{train}}(\widehat{f}_{\text{train}} = f) \tag{35}$$

The RHS of this equation is equal to the RHS of what we are showing (as shown with these previous steps so lets substitute it back in as the following,

$$\sum_{f \in \mathcal{F}} \mathbb{E}_{\text{train}}[\widehat{\epsilon}_{\text{train}}(\widehat{f}_{\text{train}})]\mathbb{P}_{\text{train}}(\widehat{f}_{\text{train}} = f) \leq \mathbb{E}_{\text{train,test}}[\widehat{\epsilon}_{\text{test}}(\widehat{f}_{\text{train}})] \tag{36}$$

The LHS is now the expectation of the expectation and therefore it simplifies to the following,

$$\mathbb{E}_{\text{train}}[\widehat{\epsilon}_{\text{train}}(\widehat{f}_{\text{train}})] \leq \mathbb{E}_{\text{train,test}}[\widehat{\epsilon}_{\text{test}}(\widehat{f}_{\text{train}})] \tag{37}$$

As expected.

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 (see Figure 1).

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?

Solution. This model is binning the points into sections of size m and then computing the expected value of each bin. Therefore, we expect that for small values of m (few bins) that the bias will be high and the variance will be low since the model is simpler. For large values of m, we expect the bias to be low since the expected value is computed over a smaller bin but we expect the variance to be high since the model is now more complex.

---

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

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

Solution. To show that this is true, lets first think of what this is representing. The LHS is the average bias-squared and the RHS is also showing the average bias-squared but as a sum over each partitioned section of the model. To start this, lets first look at the LHS of the equation and recall that any summation can be decomposed into a summation of summations. In particular we want to decompose the LHS into a summation over each interval of size m. We will write this as the following,

$$\frac{1}{n}\sum_{i=1}^{n}(\mathbb{E}[\widehat{f}_m(x_i)] - f(x_i))^2 = \frac{1}{n}\sum_{j=1}^{n/m}\sum_{i=(j-1)m+1}^{jm}(\mathbb{E}[\widehat{f}_m(x_i)] - f(x_i))^2 \tag{38}$$

From here, lets look at an example such as when we are in the first bin, j=1, and in this example we know that the variable x_i is within the interval $x \in \left(\frac{(j-1)m}{n}, \frac{jm}{n}\right]$ by definition in our example and therefore the indicator function is unity here, and substituting the definition of y_i results in the following,

$$\frac{1}{n}\sum_{j=1}^{n/m}\sum_{i=(j-1)m+1}^{jm}(\mathbb{E}[\widehat{f}_m(x_i)] - f(x_i))^2 = \frac{1}{n}\sum_{j=1}^{n/m}\sum_{i=(j-1)m+1}^{jm}(\mathbb{E}[\frac{1}{m}\sum_{i=1}^{m} f(x_i)] - f(x_i))^2 \tag{39}$$

We now use the definition of $\bar{f}^{(1)} = \frac{1}{m}\sum_{i=1}^{m} f(x_i)$ for the first bin and not that since we picked and an x_i in the first bin, $\bar{f}^{(1)}$ is now a constant so the expectation of a constant is just the constant leaving the following equation for the first bin.

$$= \frac{1}{n}\sum_{i=1}^{m}(\bar{f}^{(1)} - f(x_i))^2 \tag{40}$$

We can now see that this pattern will exist for each bin and since we wan to sum over all bins we can rewrite this as the following.

$$= \frac{1}{n}\sum_{j=1}^{n/m}\sum_{i=(j-1)m+1}^{jm}(\bar{f}^{(j)} - f(x_i))^2 \tag{41}$$

This is now the RHS of what we are trying to show, as expected.

---

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)

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

Solution. Let's start with the furthest left side and again we know that we can write any summation as a sum of summations and so we will write the summation over all, n, values as the sum over bins and the inner sum is the sum over all values within the bin as the following.

$$\mathbb{E}\left[\frac{1}{n}\sum_{i=1}^{n}(\widehat{f}_m(x_i) - \mathbb{E}[\widehat{f}_m(x_i)])^2\right] = \mathbb{E}\left[\frac{1}{n}\sum_{j=1}^{n/m}\sum_{i=(j-1)m+1}^{jm}(\widehat{f}_m(x_i) - \mathbb{E}[\widehat{f}_m(x_i)])^2\right] \tag{42}$$

Using the same logic from part A, we know that $\mathbb{E}\left[\widehat{f}_m(x_i)\right] = \bar{f}^{(j)}$, substituting this in we get the following,

$$= \mathbb{E}\left[\frac{1}{n}\sum_{j=1}^{n/m}\sum_{i=(j-1)m+1}^{jm}(\widehat{f}_m(x_i) - \bar{f}^{(j)})^2\right] \tag{43}$$

We will now substitute the definition of $\widehat{f}_m(x_i)$ within a specific bin, j, which is we are in a specific bin, j, then the indicator function must be unity and it reduces to just c_j which results in the following.

$$= \mathbb{E}\left[\frac{1}{n}\sum_{j=1}^{n/m}\sum_{i=(j-1)m+1}^{jm}(c_j - \bar{f}^{(j)})^2\right] \tag{44}$$

Due to the linearity of expectation we can bring it inside of the summations as the following,

$$= \frac{1}{n} \sum_{j=1}^{n/m} \mathbb{E} \left[\sum_{i=(j-1)m+1}^{jm} (c_j - \bar{f}^{(j)})^2 \right] \tag{45}$$

we now see that the expression that is being summed over is no longer dependent on the index i and therefore that summation becomes m for each bin resulting in the following.

$$= \frac{1}{n} \sum_{j=1}^{n/m} m\mathbb{E} \left[(c_j - \bar{f}^{(j)})^2 \right] \tag{46}$$

Which is equal to the first equality in that we are showing. To show the second equality we need to look at c_j and we will substitute the definition into the expression about,

$$= \frac{1}{n} \sum_{j=1}^{n/m} m\mathbb{E} \left[(\frac{1}{m} \sum_{i=(j-1)m+1}^{jm} y_i - \bar{f}^{(j)})^2 \right] \tag{47}$$

Now substituting the definition of y_i we get the following,

$$= \frac{1}{n} \sum_{j=1}^{n/m} m\mathbb{E} \left[(\frac{1}{m} \sum_{i=(j-1)m+1}^{jm} (f(x_i) + \epsilon_i) - \bar{f}^{(j)})^2 \right] \tag{48}$$

Distribution the summation over $f(x_i)$ and ϵ_i we see that the first term is identically $\bar{f}^{(j)}$ which simplifies this entire expression to,

$$= \frac{1}{n} \sum_{j=1}^{n/m} m\mathbb{E} \left[(\frac{1}{m} \sum_{i=(j-1)m+1}^{jm} (\epsilon_i))^2 \right] \tag{49}$$

Rearranging this and pulling the expectation inside of the summation we find the following,

$$= \frac{1}{n} \sum_{j=1}^{n/m} m\frac{1}{m^2} (\sum_{i=(j-1)m+1}^{jm} \mathbb{E} \left[(\epsilon_i))^2 \right] \tag{50}$$

The expectation of the gaussian noise squared is by definition the variance and therefore this simplifies to,

$$= \frac{1}{n} \sum_{j=1}^{n/m} \frac{1}{m} (\sum_{i=(j-1)m+1}^{jm} \sigma^2) \tag{51}$$

Now all indices have been contracted and so the summation simplify to the following.

$$= \frac{1}{n} \frac{n}{m} \frac{1}{m} (m\sigma^2) \tag{52}$$

Which simplifying more becomes,

$$= \frac{\sigma^2}{m} \tag{53}$$

As expected for the second equality.

---

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. Lets first show that the average bias-squared is $O(\frac{L^2m^2}{n^2})$. To do this we will first consider the Lipschitz condition for the maximum and minimum indices within any given bin and therefore we know the following.

$$||f(x_{\text{i,max}}) - f(x_{\text{i,min}})|| \leq \frac{L}{n}||i_{\text{max}} - i_{\text{min}}|| \tag{54}$$

We now notice that the RHS of this inequality is the size of the bin, m, and recalling the mean-value theorem we know that $\bar{f}^{(j)}$ must be between $f(x_{\text{i,max}})$ and $f(x_{\text{i,min}})$ and therefore the following must be true,

$$||\bar{f}^{(j)} - f(x_{\text{i,min}})|| \leq \frac{Lm}{n} \tag{55}$$

Now substituting this into the RHS of the equation we showed in part b, we find the following,

$$\frac{1}{n}\sum_{j=1}^{n/m}\sum_{i=(j-1)m+1}^{jm}(\bar{f}^{(j)} - f(x_i))^2 \leq \frac{1}{n}\sum_{j=1}^{n/m}\sum_{i=(j-1)m+1}^{jm}(\frac{Lm}{n})^2 \tag{56}$$

Now we see that the summations are independent of the quantity within in parentheses and so this sums up to the following,

$$\frac{1}{n}\sum_{j=1}^{n/m}\sum_{i=(j-1)m+1}^{jm}(\bar{f}^{(j)} - f(x_i))^2 \leq \frac{1}{n}\frac{n}{m}m\frac{L^2m^2}{n^2} \tag{57}$$

$$\leq O(\frac{L^2m^2}{n^2}) \tag{58}$$

Which is what we expect. We now need to derive the minimal error with respect to m. To do this we will use the expression above and take the derivative with respect to m, set it equal to zero and solve for m. This is done in the following steps,

$$0 = \frac{\partial}{\partial m}[\frac{L^2m^2}{n^2} + \frac{\sigma^2}{m}] \tag{59}$$

$$= \frac{2L^2m}{n^2} - \frac{\sigma^2}{m^2} \tag{60}$$

$$m = (\frac{\sigma^2n^2}{2L^2})^{1/3} \tag{61}$$

Now that we have solved for the value of m, we will substitute it back into the expression for the

total error then simplify as the following,

$$= O(\frac{L^2(\frac{\sigma^2 n^2}{2L^2})^{2/3}}{n^2} + \frac{\sigma^2}{(\frac{\sigma^2 n^2}{2L^2})^{1/3}}) \tag{62}$$

$$= \frac{3}{2^{2/3}}(\frac{L\sigma^2}{n})^{2/3} \tag{63}$$

With this form, we see that as the variance of the Gaussian noise increases, we expect to see more error in the model which is expected. We also see that as the number of data points increases the error decreases which also is intuitively expected. The parameter, L, relates how much smaller the difference between the model output of from two points is to the difference between those two points which is equivalently a statement of the variance of the model so as the variance increases, we expect that the total erorr will increase as well.

---

[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

$$h_{\boldsymbol{\theta}}(x) = \theta_0 + \theta_1 \phi_1(x) + \theta_2 \phi_2(x) + \ldots + \theta_d \phi_d(x) \ , \tag{64}$$

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 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.

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.

Solution. The polynomial regression was implemented as the following and has been submitted to gradescope as well. The first part of the implementation is the constructor of the model object as the following.

```
    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
```

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

```
        # Fill in with matrix with the correct shape
        self.weight: np.ndarray = np.empty(degree)  # type: np.ndarray (degree, 1)
        # You can add additional fields
        self.mean_standard = np.empty(degree)
        self.std_standard = np.empty(degree)

```

In this, I added an empty array for the weights that will be learned and empty arrays for the mean and standard deviations used for standardizing the dataset based on the training data. Following this, I implemented the polynomial basis expansion in the poly features function below.

```
    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.

        """
        X_expanded = np.empty((X.size, degree))
        for j in range(degree):
            X_expanded[:,j] = np.squeeze(np.power(X, j+1))

        return X_expanded

```

We then implement the fit function that uses the closed form solution for the polynomial regression problem to compute the weights of the model given a training dataset. The implementation the fit function is shown below.

```
    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.
        """
        # number of data points
        n = X.size

        # polynomial expansion of each point
        X_expanded = self.polyfeatures(X, self.degree)

        # standardize the polynomial expanded data
        self.mean_standard = np.mean(X_expanded, axis=0)
        self.std_standard = np.std(X_expanded, axis=0)
        X_expanded_standard = (X_expanded - self.mean_standard)/self.std_standard

        # add 1s column
        X_ = np.c_[np.ones([n, 1]), X_expanded_standard]
        _, degree_with_constant = X_.shape

        # construct reg matrix
        reg_matrix = self.reg_lambda * np.eye(degree_with_constant)
        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)
```

```

```

We then use the trained weights for the model in the predict function to predict the output values from the polynomial regression and the implementation is shown in the code below.

```
    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.
        """
        # number of data points
        n = X.size

        # polynomial expansion of each point
        X_expanded = self.polyfeatures(X, self.degree)

        # standardize the polynomial expanded data
        X_expanded_standard = (X_expanded - self.mean_standard)/self.std_standard

        # add 1s column
        X_ = np.c_[np.ones([n, 1]), X_expanded_standard]

        # predict
        return X_.dot(self.weight)

```

Note that since we are using a polynomial expansion, we needed to standardize the data for each so that the weights are not badly scaled and therefore when we standardize the training data, we need to apply the same standardization to the data when we predict with it as shown in the code above.

---

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).

Solution. The effect of increasing the regularization is reducing the complexity of the model. Lets first look at the regression with a regularization of $\lambda = 0$ as in Figure 2.

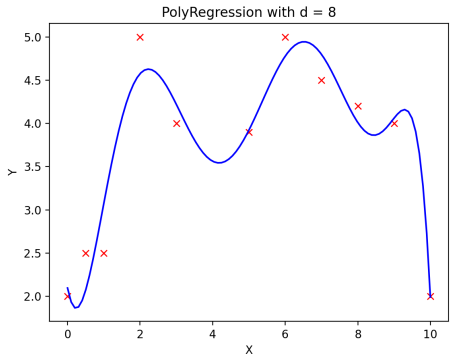

Figure 2: Polynomial regression for the given dataset with a regularization of zero.

Now if we increase the regularization, we see that the model becomes simpler and generalizes better to the data as in Figure 3.

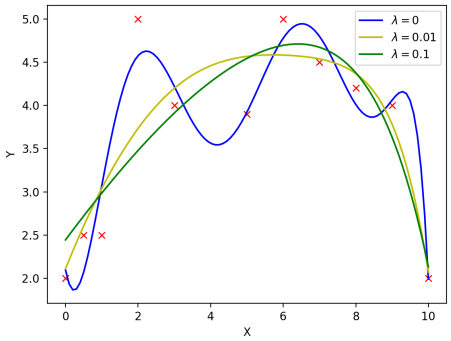

Figure 3: Polynomial regression for the given dataset with varying regularizations from 0, 0.01, and 0.1.

We see that changing the regularization by a little bit drastically changes the model but increasing more past that doesn't simplify the model much more.

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

$$\frac{1}{n}\sum_{i=1}^{n}(h_{\boldsymbol{\theta}}(\mathbf{x}_i) - y_i)^2 \ . \tag{65}$$

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:
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. In order to compute the learning curves on the polynomial regression we use the function learningCurve. This function takes in a training and test dataset along with a polynomial degree and a regularization value. These are then used to compute the training and testing error as a function of the number of samples used. The code to compute the learning curves is shown below.

```
    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)

        # instantiate model
        model = PolynomialRegression(degree=degree, reg_lambda=reg_lambda)
        errorTrain = np.zeros(n)
        errorTest = np.zeros(n)
        for i in range(n):
            model.fit(Xtrain[0:(i+1)], Ytrain[0:(i+1)])
            errorTrain[i] = mean_squared_error(model.predict(Xtrain[0:(i+1)]), Ytrain[0:(i+1)])
            errorTest[i] = mean_squared_error(model.predict(Xtest[0:(i+1)]), Ytest[0:(i+1)])

    return [errorTrain, errorTest]

```

We then apply this function to a dataset and compute the learning curves as a function of the degree and the regularization parameter as shown in Figure 4. Here we see that in general the training error converges faster than the testing error and we see that for high regularizaton values, the testing error also seems to reduce quickly.

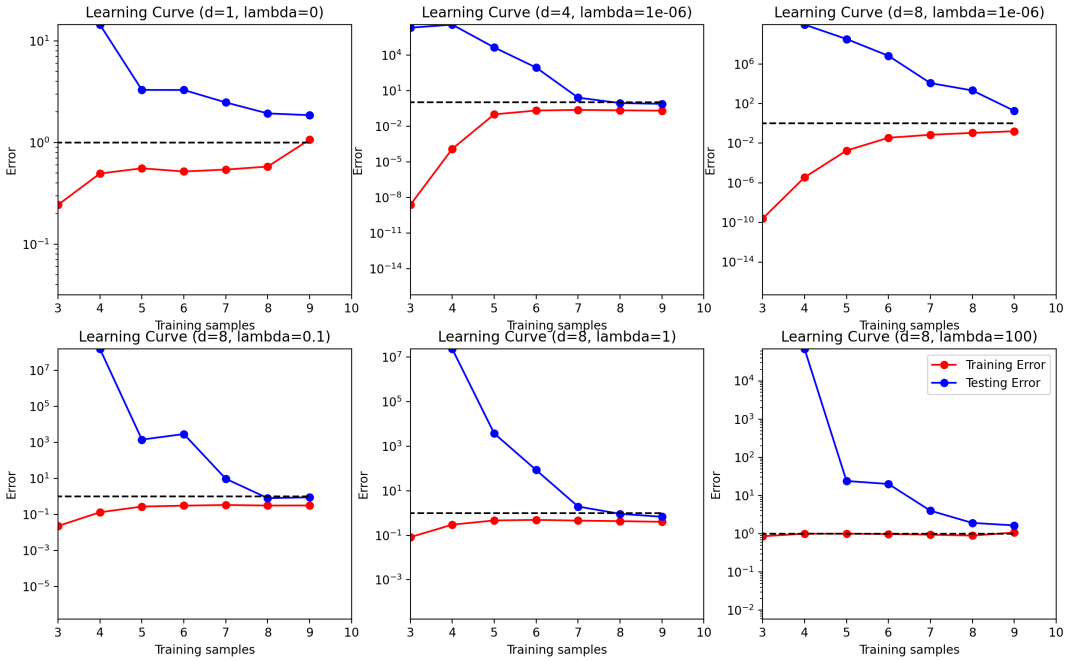

Figure 4: Learning curves for various values of d and λ. The blue lines represent the testing error, while the red lines the training error.

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`).

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 **??** 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

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

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

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

Solution.
Lets first start by writing the simplified expression given above as the following,

$$\sum_{j=1}^{k}\left[\|Xw_j - Ye_j\|^2 + \lambda\|w_j\|^2\right] = \sum_{j=1}^{k}[\sum_{i=1}^{n}(\sum_{m=1}^{d} x[i][m]w_j[m] - \sum_{j=1}^{d} y[i][j]e_j[j]) + \lambda\sum_{m=1}^{d}(w_j[m]^2)] \tag{66}$$

We now want to take the gradient with respect to the vector w_j and set that equal to zero to find the maximum likelihood estimator as the following.

$$0 = \frac{\partial}{\partial w_j[l]}\sum_{j=1}^{k}[\sum_{i=1}^{n}(\sum_{m=1}^{d} x[i][m]w_j[m] - \sum_{j=1}^{d} y[i][j]e_j[j]) + \lambda\sum_{m=1}^{d}(w_j[m]^2)] \tag{67}$$

Since the gradient is a linear operator, we can rearrange this to be the following,

$$0 = \sum_{j=1}^{k}[\sum_{i=1}^{n}\frac{\partial}{\partial w_j[l]}(\sum_{m=1}^{d} x[i][m]w_j[m] - \sum_{j=1}^{d} y[i][j]e_j[j]) + \lambda\sum_{m=1}^{d}(w_j[m]^2)] \tag{68}$$

Taking the gradient of these sums we get the following,

$$0 = \sum_{j=1}^{k}[\sum_{i=1}^{n} 2(x[i][l]w_j[l] - y[i])x[i][l] + 2\lambda w_j[l]] \tag{69}$$

We now divide out the 2 and we will write this is matrix notation as the following,

$$0 = X^TXw_j - X^Ty + \lambda Iw_j \tag{70}$$

Now rearranging this and solving for w_j we get the following.

$$X^Ty = X^TXw_j + \lambda Iw_j \tag{71}$$

$$\hat{w_j} = (X^TX + \lambda I)^{-1}X^Ty \tag{72}$$

$$\hat{W} = (X^TX + \lambda I)^{-1}X^TY \tag{73}$$

As expected, this is the maximum likelihood estimator.

---

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 $\arg\max_{j=0,...,9} e_j^TW^Tx_i'$ 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.

Solution.
First we implement the function train which take the training data and solves for the maximum likelihood estimator weights using using the closed form solution from part A and is shown below.

```
        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
        x_transpose = np.transpose(x)

        # solve the closed form solution for ridge regression
        weight_mapping = np.dot(x_transpose, x) + (_lambda * np.eye(d))
        weight_matrix = np.linalg.solve(weight_mapping, np.dot(x_transpose, y))

        return weight_matrix

```

Following the implementation of the training function, we implement the one_hot function that takes an input of the y labels and the number of classes in the dataset and creates an encoded matrix of the labels, the function is shown in the following.

```
        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 = y.size
        one_hot_rep = np.zeros((n, num_classes))
        for i in range(n):
            one_hot_rep[i, y[i]] = 1

        return one_hot_rep

```

After the one hot encoding function, we implement the predict function that uses the weights solved for in the train function and predicts the labels for a given dataset and the implementation is shown below.

```
        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)`.
        """
        d, k = w.shape
        n, d = x.shape

        # Create one hot encoding matrix
        e_j = np.eye(k)

        prediction_classes = np.empty(n, dtype=int)
        for i in range(n):
            prediction = np.matmul(np.transpose(w), x[i, :])
            prediction_classes[i] = int(np.argmax(np.matmul(e_j, prediction)))

        return prediction_classes

```

The main function then uses each of these function to train a model on a training dataset and then predict the labels of that dataset. We then choose some of the images that were incorrectly classified and plot them to see if we can see any patterns, discussion of this plot is in part d and code is shown below.

```
        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}%"
)

        # Find 10 images that are misclassified and plot
        incorrect_inds = ~np.equal(y_test_pred, y_test)
        incorrect_images = x_test[incorrect_inds, :]
        incorrect_labels = y_test_pred[incorrect_inds]
        incorrect_images_example = incorrect_images[:10, :].reshape((10, 28, 28))
        incorrect_labels_example = incorrect_labels[:10]
        print(incorrect_labels_example)
        fig, ax = plt.subplots(ncols=5, nrows=2)
        n = 0
        for i in range(2):
            for j in range(5):
                ax[i,j].imshow(incorrect_images_example[n,:])
```

```
                ax[i,j].set_title(f'Predicted label = {incorrect_labels_example[n]}')
                n +=1
        plt.show()

```

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

Solution.
The training and testing errors from this classier are the following:
Train Error: 14.805%
Test Error: 14.66%
This is a bit unexpected since the training error is slightly larger than the test error but they are very close. However, in general we expect the training error to be smaller than the test error.

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?

Solution.
When we use matplotlibs imshow package and plot 10 random images that were incorrectly classified we see the following in Figure 5.

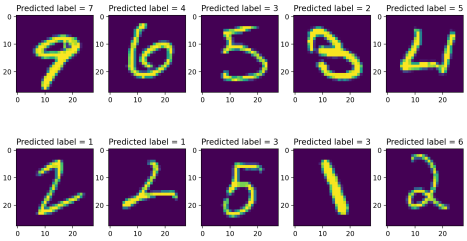

Figure 5: Incorrectly predicted integers from the MNIST dataset.

Here we see that some common integers that are mis-classified are five and two. It seem that these could be mis-classified sine they are both integers that do not have strong defining lines but are rather fluid and depending on who writes it can look more fluid or more rigid which could be challenging to define the features.

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!)

Administrative

A6.

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

Solution. In total, this homework took me about 25 hours with time spent at office hours, playing with the code and learning the material.