Homework #1

Spring 2020, CSE 446/546: Machine Learning
Richy Yun
Due: 4/24/20 11:59 PM

Short Answer and "True or False" Conceptual questions

A.0

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

  Bias can be thought of the error between the average output of the model and the real data (ground "truth"). Variance is the variability in the output of the model. Bias-variance tradeoff refers to how models with high bias result in low variance and vice versa, which forces designs to find the proper balance between bias and variance. In essence, we have to decide how well the model fits the train data - the better the fit the lower the variance, but generally higher bias due to overfitting. On the flip side, the more generalized the fit the higher the variance, but lower bias.

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

  As model complexity increases, bias goes down as the result becomes more and more overfitted to the train data, but variance goes up. As complexity decreases, bias goes up due to generalization but variance goes down.

- *[1 points]* True or False: The bias of a model increases as the amount of training data available increases.

  False. The bias decreases, as more training data results in more overfitting.

- *[1 points]* True or False: The variance of a model decreases as the amount of training data available increases.

  False. As there is more training data the model is more overfitted, leading to less generalization and high variance of the model.

- *[1 points]* True or False: A learning algorithm will generalize better if we use less features to represent our data.

  True. More features leads to overfitting which does not generalize well. Thus less features is a more generalized model.

- *[2 points]* To get better generalization, should we use the train set or the test set to tune our hyperparameters?

  Assuming that the train set is larger than the test set, we should use the test set to tune our hyperparameters, as using the smaller set will result in higher generalization.

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

  False. The training error is generally smaller than the test error (an estimate of true error) as the function is overfit to the training set. The expected value of the training error is always less than or equal to the expected value of the test error.

Maximum Liklihood Estimation (MLE)

A.1

a. *[5 points]* To derive the maximum-likelihood estimate by taking the log of the likelihood to covert it from a product to a sum, then taking the derivative and setting it equal to zero:

$$\begin{aligned}
L(\lambda) &= \prod_{i=1}^{n} \text{Poi}(x_i|\lambda) \\
&= \sum_{i=1}^{n} \log(\text{Poi}(x_i|\lambda)) \\
\hat{\lambda}_{MLE} &= \arg\max_x \ (L(\lambda)) \\
&= \arg\max_x \ \sum_{i=1}^{n} \log(e^{-\lambda}\frac{\lambda_i^x}{x_i!}) \\
0 &= \sum_{i=1}^{n} \left(\frac{d}{d\lambda}\log(e^{-\lambda}\frac{\lambda_i^x}{x_i!})\right) \\
&= \sum_{i=1}^{n} \left(\frac{d}{d\lambda}(-\lambda + x_i\log(\lambda) - \log(x_i!))\right) \\
&= \sum_{i=1}^{n} \left(-1 + \frac{x_i}{\lambda}\right) \\
n\lambda &= \sum_{i=1}^{n} x_i \\
\hat{\lambda}_{MLE} &= \frac{1}{n}\sum_{i=1}^{n} x_i \\
&= \frac{1}{5}\sum_{i=1}^{5} x_i
\end{aligned}$$

Thus, the maximum likelihood estimate is the average of the scores.

b. *[5 points]* Similar to above, the maximum likelihood estimate is the average:

$$\hat{\lambda}_{MLE} = \frac{1}{6}\sum_{i=1}^{6} x_i$$

c. *[5 points]* For 5 and 6 games, we simply find the average of the goal counts to get the estimate of λ. For 5 games, $\lambda = \frac{6}{5}$. For 6 games, $\lambda = \frac{10}{6} = \frac{5}{3}$

A.2 *[10 points]* We first approach this problem similar to part a:

$$= \begin{cases} L(\theta) = \prod_{i=1}^{n} \text{Uniform}(x_i|\theta) & 0 \leq x_i \leq \theta \\ 0 & \text{otherwise} \end{cases}$$

$$= \sum_{i=1}^{n} \log(\text{Uniform}(x_i|\theta))$$

$$\hat{\theta}_{MLE} = \arg\max_x \left(L(\theta)\right)$$

$$= \arg\max_x \sum_{i=1}^{n} \log\left(\frac{1}{\theta}\right)$$

$$0 = \sum_{i=1}^{n} \left(\frac{d}{d\theta}(\log(1) - \log(\theta))\right)$$

$$= \sum_{i=1}^{n} -\frac{1}{\theta}$$

$$= -\frac{n}{\theta}$$

However, there is no global maxima. Thus we need to determine θ by evaluating the likelihood conceptually:

$$L(\theta) = \prod_{i=1}^{n} \text{Uniform}(x_i|\theta)$$

$$= \frac{1}{\theta^n}$$

To maximize $L(\theta)$ we clearly need to minimize θ. However, $\theta \geq x_i$ for all x_i as we cannot have sampled outside the limits of the distribution. Thus, the value that satisfies both conditions is the maximum of all samples, or:

$$\hat{\theta}_{MLE} = \max(x_i)$$

Overfitting

A.3

a. *[3 points]*

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

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

Since each sample (x, y) in S is i.i.d. we can remove the sum and the $1/N_{\text{train}}$. Since f is fixed and not dependent on the training set (we haven't see any data yet), the expected value does not have to be over the train either. We thus have:

$$\mathbb{E}_{\text{train}}[\hat{\epsilon}_{\text{train}}(f)] = \mathbb{E}[(f(x) - y)^2]$$

Using the exact same logic on $\mathbb{E}_{\text{test}}[\hat{\epsilon}_{\text{test}}(f)]$ we get:

$$\mathbb{E}_{\text{train}}[\hat{\epsilon}_{\text{train}}(f)] = \mathbb{E}_{\text{test}}[\hat{\epsilon}_{\text{test}}(f)] = \epsilon(f) = \mathbb{E}[(f(x) - y)^2]$$

And using the same reasoning, since $\hat{f}$ is not dependent on the testing set, we can also show:

$$\mathbb{E}_{\text{test}}[\hat{\epsilon}_{\text{test}}(\hat{f})] = \epsilon(\hat{f}) = \mathbb{E}[(\hat{f}(x) - y)^2]$$

b. *[4 points]* No, because $\hat{f}$ is dependent on the training set, so (x,y) are not random variables anymore. Thus $\mathbb{E}_{\text{train}}[(\hat{f}(x)-y)^2] \neq \mathbb{E}[(\hat{f}(x)-y)^2]$

c. *[8 points]* From the hint, we can see that the expression is nonzero only when $f = \hat{f}_{\text{train}}$. Therefore:

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

We also know by definition that:

$$\hat{\epsilon}_{train}(\hat{f}_{train}) = \min\ \hat{\epsilon}_{train}(f)$$

Then it follows:

$$\mathbb{E}_{\text{train}}[\hat{\epsilon}_{\text{train}}(\hat{f}_{\text{train}})] = \mathbb{E}_{\text{train}}[\min\ \hat{\epsilon}_{\text{train}}(f)] \leq \min\ \mathbb{E}_{\text{train}}[\hat{\epsilon}_{\text{train}}(f)]$$

From part a, $\mathbb{E}_{\text{train}}[\hat{\epsilon}_{\text{train}}(f)] = \mathbb{E}_{\text{test}}[\hat{\epsilon}_{\text{test}}(f)]$. Thus:

$$\mathbb{E}_{\text{train}}[\hat{\epsilon}_{\text{train}}(\hat{f}_{\text{train}})] \leq \min\ \mathbb{E}_{\text{test}}[\hat{\epsilon}_{\text{test}}(f)]$$

Since $\hat{f}_{\text{train}} \in \mathcal{F}$, we can safely assume:

$$\min\ \mathbb{E}_{\text{test}}[\hat{\epsilon}_{\text{test}}(f)] \leq \mathbb{E}_{\text{test}}[\hat{\epsilon}_{\text{test}}(\hat{f}_{\text{train}})]$$

Thus, we come to the expression:

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

Bias-Variance tradeoff

B.1

a. *[5 points]* For small m bias should go up and variance down, as we reach higher complexity and closer to the specific data points. Conversely, for large m bias should go down and variance up and we reach lower complexity.

b. *[5 points]* The sum on the left can be rewritten as:

$$\sum_{i=1}^{n} = \sum_{j=1}^{n/m} \sum_{i=(j-1)m+1}^{jm}$$

Thus, we essentially need to show that $\mathbb{E}[\hat{f}_m(x_i)] = \bar{f}^{(j)}$:

$$\mathbb{E}[\hat{f}_m(x_i)] = \mathbb{E}\left[\sum_{j=1}^{n/m} c_j \mathbf{1}\{x_i \in \left(\frac{(j-1)m}{n}, \frac{jm}{m}\right]\}\right]$$

For any one x_i, the $\mathbf{1}$ function is only true for one j. Therefore, we can remove the summation and the $\mathbf{1}$ function:

$$\begin{aligned}
\mathbb{E}[\hat{f}_m(x_i)] &= \mathbb{E}[c_j] \\
&= \mathbb{E}\left[\frac{1}{m}\sum_{i=(j-1)m+1}^{jm} y_i\right] \\
&= \mathbb{E}\left[\frac{1}{m}\sum_{i=(j-1)m+1}^{jm} (f(x_i)+\epsilon_i)\right] \\
&= \mathbb{E}\left[\frac{1}{m}\sum_{i=(j-1)m+1}^{jm} f(x_i)\right]
\end{aligned}$$

With the last step because we know ϵ_i is normally distributed with mean of 0. Since c_j is already calculating the average, the expected value is itself. Thus: $\mathbb{E}[\hat{f}_m(x_i)] = \bar{f}^{(j)}$ and the average bais-squared equation holds true.

c. *[5 points]* The left most sum can be rewritten as:

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

Then, we can rewrite the sum as:

$$\sum_{i=1}^{n} = \sum_{j=1}^{n/m} m$$

We can see that the internal expected value term is the variance of $\hat{f}_m(x_i)$. For the second term, we can clearly see the expected value term, $\mathbb{E}[(c_j - \bar{f}^{(j)})^2]$ is simply the variance of c_j as $\mathbb{E}[c_j] = \bar{f}^{(j)}$. Thus, we have to show that the two variances are equal. The variance of $\hat{f}_m(x_i)$ is:

$$\text{var}[\hat{f}_m(x_i)] = \text{var}\left[\sum_{j=1}^{n/m} c_j \mathbf{1}\{x_i \in \left(\frac{(j-1)m}{n}, \frac{jm}{m}\right]\}\right]$$

Using the same logic as part b, we can see that for a single term x_i, $\mathbf{1}$ holds true for only one j, so the sum and the function can be removed as before. Thus:

$$\text{var}[\hat{f}_m(x_i)] = \text{var}[c_j]$$

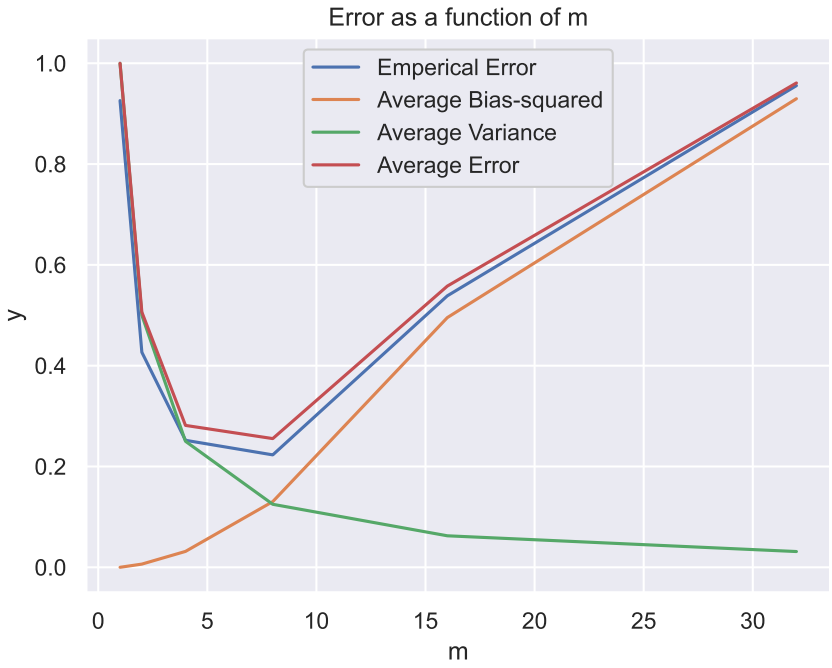
which resolves the first equality. Now we can look at the variance of c_j to simplify the second term:

$$\begin{aligned}
\text{var}[c_j] &= \text{var}\left[\frac{1}{m}\sum_{i=(j-1)m+1}^{jm} y_i\right] \\
&= \text{var}\left[\frac{1}{m}\sum_{i=(j-1)m+1}^{jm} (f(x_i) + \epsilon_i)\right] \\
&= \text{var}\left[\frac{1}{m}\sum_{i=(j-1)m+1}^{jm} f(x_i)\right] + \text{var}\left[\frac{1}{m}\sum_{i=(j-1)m+1}^{jm} \epsilon_i\right] \\
&= \text{var}\left[\frac{1}{m}\sum_{i=(j-1)m+1}^{jm} \epsilon_i\right] \\
&= \frac{1}{m^2} \times m\sigma^2 \\
&= \frac{\sigma^2}{m}
\end{aligned}$$

$f(x)$ and ϵ can be separated as they are independent. The sum with $f(x_i)$ is essentially saying $\text{var}[\bar{f}^{(j)}] = \text{var}[\mathbb{E}[\hat{f}_m(x_i)]]$, and the variance of an expected value is zero. Now we can plug that into the original expression:

$$\begin{aligned}
\frac{1}{n}\sum_{j=1}^{n/m} m\mathbb{E}[(c_j - \bar{f}^{(j)})^2] &= \frac{1}{n}\sum_{j=1}^{n/m} m\frac{\sigma^2}{m} \\
&= \frac{1}{n}\frac{n}{m}m\frac{\sigma^2}{m} \\
&= \frac{\sigma^2}{m}
\end{aligned}$$

d. *[15 points]* As expected, increasing m results in a higher bias but lower variance, as larger m means lower complexity. The spot that gives the least error of the tested values is when $m = 8$, giving a clear example of bias-variance tradeoff.



e. *[5 points]* The expression given by the Mean-Value theorem allows us to treat $\bar{f}^{(j)}$ as some $f(x_k)$ where $(j-1)m+1 \leq k \leq jm$. Therefore, the inside of the average bias squared expression, $\bar{f}^{(j)} - f(x_i)$ can be thought of as $f(x_k) - f(x_i)$. The L-Lipschitz property of f then tells us that difference is governed by $\frac{L}{n}|k-i|$. However, $k-i$ is governed by m and averages to be $\frac{m}{2}$, or $O(m)$. Thus, the term $\bar{f}^{(j)} - f(x_i)$ is $O(\frac{Lm}{n})$. Applying it to the definition of average bias-squared, we get:

$$\begin{aligned} O(\text{averag-bias squared}) &= \frac{1}{n}\sum_{j=1}^{n/m}\sum_{i=(j-1)m+1}^{jm}(\frac{Lm}{n})^2 \\ &= \frac{1}{n}\frac{n}{m}m\frac{L^2m^2}{n^2} \\ &= \frac{L^2m^2}{n^2} \end{aligned}$$

To minimize the expression for total error, we take the derivative and set it equal to zero:

$$\begin{aligned} \frac{d}{dm}\left(\frac{L^2m^2}{n^2}+\frac{\sigma^2}{m}\right) = \frac{2L^2m}{n^2} - \frac{\sigma^2}{m^2} &= 0 \\ 2L^2m^3 - n^2 &= 0 \\ m &= \left(\frac{n^2\sigma^2}{2L^2}\right)^{1/3} \end{aligned}$$

Plugging it back into the total error expression, we get:

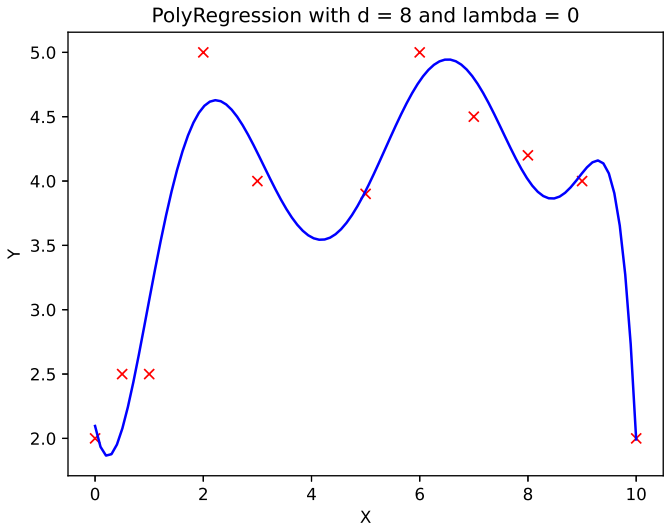
$$O(\text{Error}) = \frac{L^2m^2}{n^2} + \frac{\sigma^2}{m} = \frac{L^2}{n^2}\left(\frac{n^2\sigma^2}{2L^2}\right)^{1/3} + \sigma^2\left(\frac{2L^2}{n^2\sigma^2}\right)^{1/3}$$

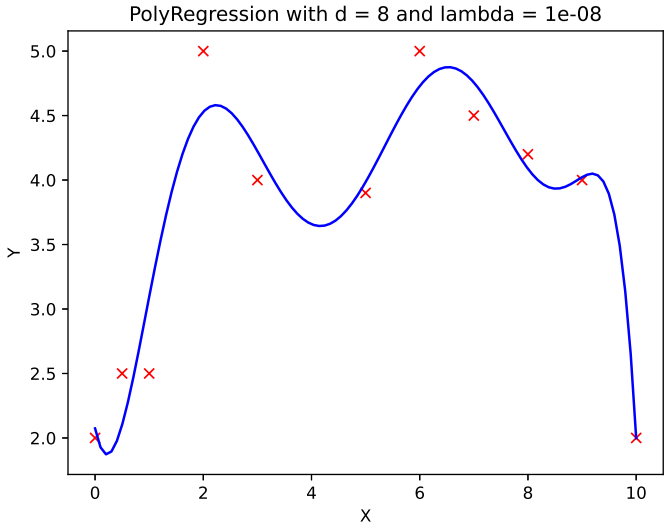
$$\begin{aligned} O(\text{Error})^3 &= \frac{L^6}{n^6}\frac{n^2\sigma^2}{2L^2} + \sigma^6\frac{2L^2}{n^2\sigma^2} \\ &= \frac{L^4\sigma^2}{2n^4} + \frac{2L^2\sigma^4}{n^2} \\ &= \frac{L^4\sigma^2 + 4n^2L^2\sigma^4}{2n^4} \\ O(\text{Error}) &= \left(\frac{L^4\sigma^2 + 4n^2L^2\sigma^4}{2n^4}\right)^{1/3} \end{aligned}$$

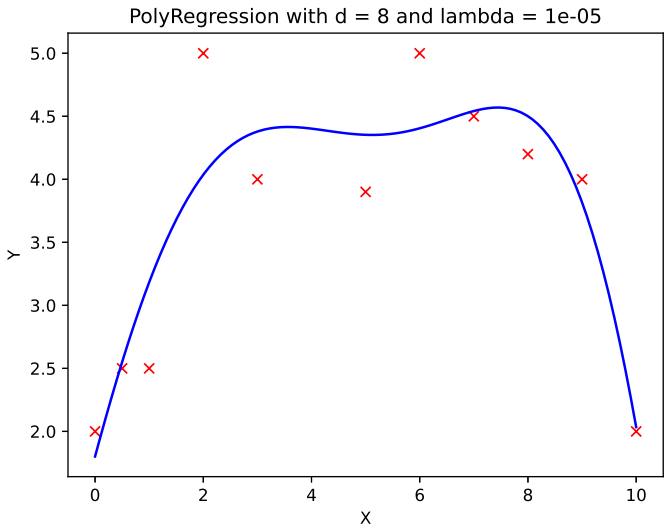
Looking at the error, we get larger error for larger σ^2, which is expected due to the increase in variance. We also get larger error for larger L but smaller error for larger n which again is expected with their relationship with the bias. With larger σ^2 we need larger m to bring the variance down (larger m is less complexity). With larger L or smaller n we need smaller m to bring the bias down. All variables behave as expected.

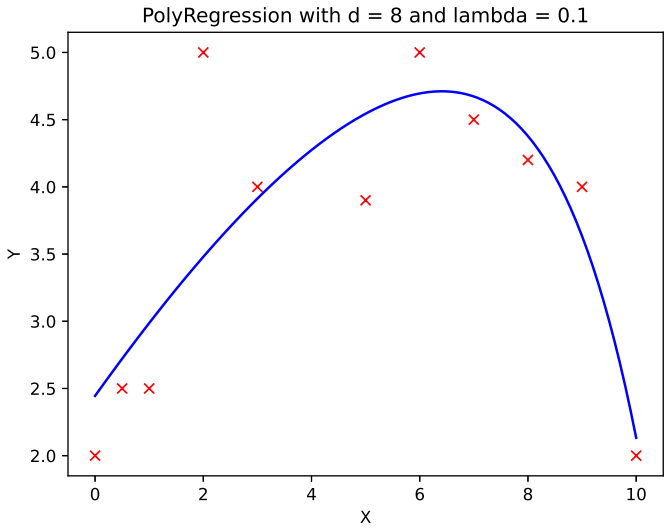
Polynomial Regression

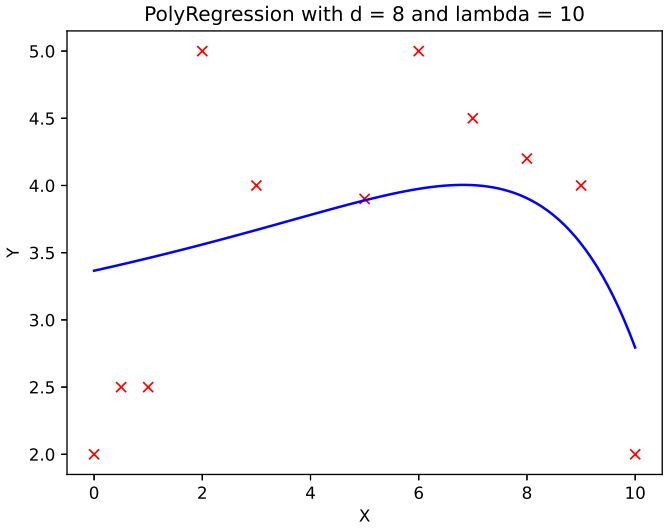
A.4 *[10 points]* As we increase regularization, we end up with less complex, more generalized solutions. We see subtle changes with λ as low as $1e-8$, with many features being lost at $1e-5$. At $\lambda = 0.1$ we essentially have a quadratic solution, with even those features becoming dimished at $\lambda = 10$.



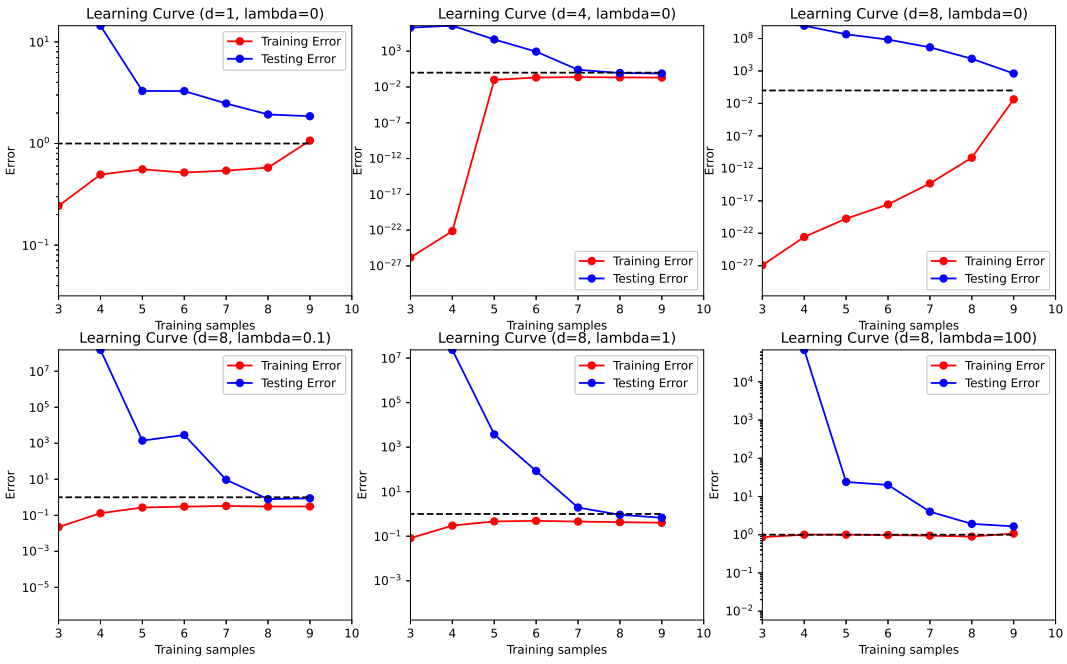

PolyRegression with d = 8 and lambda = 1e-08
5.0
4.5
4.0
3.5
3.0
2.5
2.0
Y
0
2
4
6
8
10
X


PolyRegression with d = 8 and lambda = 1e-05
5.0
4.5
4.0
3.5
3.0
2.5
2.0
Y
0
2
4
6
8
10
X


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



A.5 *[10 points]* As expected, we see greater training error with more samples as the model cannot perfectly fit all the data, which is compensated by the increased number of degrees (top row). With higher regularization we also see higher training error due to the increased generalization (bottom row). For testing error, we see an increase with the increase of degrees likely due to overfitting to the training set (top row), which is mitigated by inccreasing regularization for generalization of the model (bottom row).



A.6

a. *[10 points]* To find $\widehat{W}$ we take the gradient with respect to W and set it equal to zero to find the minimum as shown during lecture:

$$\nabla_W \left[\sum_{j=0}^{k} ||Xw_j - Ye_j||^2 + \lambda ||w_j||^2 \right] = 2X^T(XW - Y) + 2\lambda W = 0$$

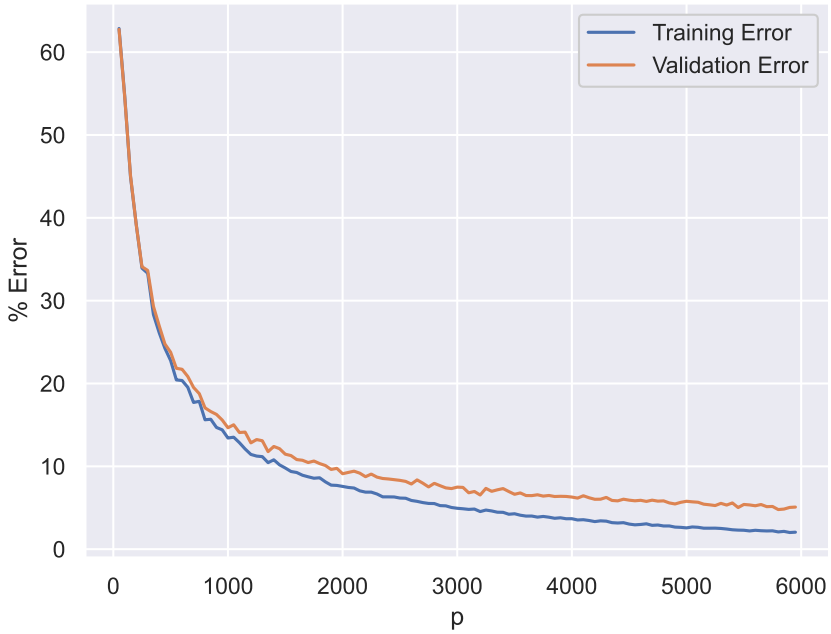
$$\begin{aligned} X^T(XW - Y) + \lambda W &= 0 \\ X^T XW - X^T Y + \lambda W &= 0 \\ X^T XW + \lambda W &= X^T Y \\ (X^T X + \lambda I)W &= X^T Y \\ \widehat{W} &= (X^T X + \lambda I)^{-1} X^T Y \end{aligned}$$

e is omitted from the calculations as it is a simple one-hot encoding, and we can assume the training and test labels (Y_{train} and Y_{test}) are definitive (i.e. already given as one-hot encoded). The identity matrix, I, is included with λ when isolating W to form a matrix and allow for the addition.

b. *[10 points]* The training error is 14.805% and the testing error is 14.66%.

B.2

a. *[10 points]* As p increases, error decreases as expected when increasing the number of features. As expected, higher p leads to a larger difference in the training and validation error as well, due to overfitting. At 6000 dimensions, the training error drops to $\sim 2\%$ and the validation error to $\sim 5\%$. Although we would expect validation error to have an absolute minima because of overfitting, we weren't able to reach the number of dimensions required for that case due to λ allowing the model to be more generalized. With a smaller training set or smaller λ the validation error would diverge faster from the training error and have an absolute minima with lower values of p.



b. *[5 points]*

Source Code - should be at the end of each problem

B.1d

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
sns.set()

# Define function
def fx(x):
    return 4*np.sin(np.pi*x)*np.cos(6*np.pi*x**2)

# Define average bias-squared
def bias(n, m, x):
        b = 0
        for j in range(1, int(n/m)+1):                #+1 so it includes int(n/m)

                fj = 0
                for k in range((j-1)*m+1, j*m+1):     #+1 for inclusion
                        fj = fj + fx(x[k-1])          #-1 for 0 based indexing
                fj = fj/m

                for i in range((j-1)*m+1, j*m+1):     #+1 for inclusion
                        b = b + (fj-fx(x[i-1]))**2    #-1 for 0 based indexing

        return b/n

# Initialize variables
n = 256
M = [1, 2, 4, 8, 16, 32]
x = np.linspace(0, 1, n)
f = fx(x)
y = fx(x) + np.random.normal(0, 1, n)

# Calculate
EmpErr = [None]*len(M)
Bias = [None]*len(M)
Var = [None]*len(M)
TotErr = [None]*len(M)
for i in range(0, len(M)):
        # Calculate fm
        m = M[i]
        bins = np.linspace(0, n, int(n/m)+1)
        inds = np.linspace(0, n-1, n)
        dig = np.digitize(inds, bins)
        fm = [y[dig == j].mean() for j in range(1, len(bins))]
        inds = np.array(inds, dtype='int')
        FM = [None]*len(inds)
        for j in range(0, len(inds)):
                FM[j] = fm[dig[j]-1]

        # Set variables for this m
        err = FM-f;
        sqerr = [e ** 2 for e in err]
        EmpErr[i] = sum(sqerr)/n
```

```
        Bias[i] = bias(n, m, x)
        Var[i] = 1/m
        TotErr[i] = Bias[i]+Var[i]

# Plot
plt.plot(M, EmpErr)
plt.plot(M, Bias)
plt.plot(M, Var)
plt.plot(M, TotErr)

plt.xlabel("m")
plt.ylabel('y')
plt.legend(['Emperical Error', 'Average Bias-squared', 'Average Variance', 'Average Error'])
plt.title('Error as a function of m')
plt.show()
```

A.4 and A.5 (polyreg.py)

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

import numpy as np

#-----------------------------------------------------------------
#  Class PolynomialRegression
#-----------------------------------------------------------------

class PolynomialRegression:

    def __init__(self, degree=1, reg_lambda=1E-8):
        """
        Constructor
        """
        self.degree = degree
        self.reg_lambda = reg_lambda
        self.theta = None
        self.avg = None      # for normalization
        self.std = None      # for normalization

    def polyfeatures(self, X, degree):
        """
        Expands the given X into an n * d array of polynomial features of
            degree d.

        Returns:
            A n-by-d numpy array, with each row comprising of
            X, X * X, X ** 3, ... up to the dth power of X.
            Note that the returned matrix will not include the zero-th power.

        Arguments:
            X is an n-by-1 column numpy array
            degree is a positive integer
```

```
        """

        # Polynomial expansion
        n = len(X)
        xpoly = np.empty([n, degree])
        for x in range(0, n):
            for d in range(0, degree):
                xpoly[x, d] = np.power(X[x], d+1)

        return xpoly

    def fit(self, X, y):
        """
            Trains the model
            Arguments:
                X is a n-by-1 array
                y is an n-by-1 array
            Returns:
                No return value
            Note:
                You need to apply polynomial expansion and scaling
                at first
        """
        # Polynomial expansion
        x = self.polyfeatures(X, self.degree)

        # Normalization
        self.avg = np.mean(x, axis=0)
        self.std = np.std(x, axis=0)
        x = x-self.avg[None, :]
        x = x/self.std[None, :]

        # From linreg_closedform.py
        ################################
        n = len(X)

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

        n, d = X_.shape
        d = d - 1  # remove 1 for the extra column of ones we added

        # construct reg matrix
        reg_matrix = self.reg_lambda * np.eye(d + 1)
        reg_matrix[0, 0] = 0

        # analytical solution (X'X + regMatrix)^-1 X' y
        self.theta = np.linalg.pinv(X_.T.dot(X_) + reg_matrix).dot(X_.T).dot(y)
        ################################

    def predict(self, X):
        """
        Use the trained model to predict values for each instance in X
        Arguments:
            X is a n-by-1 numpy array
```

```
        Returns:
            an n-by-1 numpy array of the predictions
        """

        # Polynomial expansion
        x = self.polyfeatures(X, self.degree)

        # Normalization
        x = x - self.avg[None, :]
        x = x / self.std[None, :]

        # From linreg_closedform.py
        ################################
        n = len(X)

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

        # predict
        return X_.dot(self.theta)
        ################################

#-----------------------------------------------------------------
#  End of Class PolynomialRegression
#-----------------------------------------------------------------

def learningCurve(Xtrain, Ytrain, Xtest, Ytest, reg_lambda, degree):
    """
    Compute learning curve

    Arguments:
        Xtrain -- Training X, n-by-1 matrix
        Ytrain -- Training y, n-by-1 matrix
        Xtest -- Testing X, m-by-1 matrix
        Ytest -- Testing Y, m-by-1 matrix
        regLambda -- regularization factor
        degree -- polynomial degree

    Returns:
        errorTrain -- errorTrain[i] is the training accuracy using
        model trained by Xtrain[0:(i+1)]
        errorTest -- errorTest[i] is the testing accuracy using
        model trained by Xtrain[0:(i+1)]

    Note:
        errorTrain[0:1] and errorTest[0:1] won't actually matter, since we start
        displaying the learning curve at n = 2 (or higher)
    """

    n = len(Xtrain)

    errorTrain = np.zeros(n)
    errorTest = np.zeros(n)

    reg = PolynomialRegression(degree, reg_lambda)
```

```
    for i in range(1, n):
        reg.fit(Xtrain[0:i+1], Ytrain[0:i+1])
        trainpredict = reg.predict(Xtrain[0:i+1])
        errorTrain[i] = np.sum((trainpredict - Ytrain[0:i+1])**2)/len(Ytrain[0:i+1])
        testpredict = reg.predict(Xtest)
        errorTest[i] = np.sum((testpredict - Ytest)**2)/len(Ytest)

    return errorTrain, errorTest
```

A.6b

```
import numpy as np
import matplotlib.pyplot as plt
from mnist import MNIST
from scipy import linalg

def train(X, Y, lam):
    d = int(X.size/len(X))
    reg_matrix = lam * np.eye(d)
    a = np.matmul(X.transpose(), X) + reg_matrix
    b = np.matmul(X.transpose(), Y)
    W = linalg.solve(a, b)
    return W

def predict(W, X):
    d = len(X)
    Y = np.zeros(d)
    temp = np.matmul(X, W)
    for i in range(0, d):
        ind = np.where(temp[i, :] == np.amax(temp[i, :]))
        Y[i] = ind[0]
    return Y

# Load data
mndata = MNIST('./data/')
X_train, labels_train = map(np.array, mndata.load_training())
X_test, labels_test = map(np.array, mndata.load_testing())

# Rearrange data
X_train = X_train/255.0
X_test = X_test/255.0
Y_train = np.zeros((len(X_train), 10))
Y_train[range(0, len(X_train)), labels_train] = 1
Y_test = np.zeros((len(X_test), 10))
Y_test[range(0, len(X_test)), labels_test] = 1

# Train model
lam = 0.0001
W = train(X_train, Y_train, lam)

# Get predictions
P_train = predict(W, X_train)
P_test = predict(W, X_test)
```

```
# Error calculation
train_error = (len(X_train) - np.sum(P_train == labels_train))/len(X_train)
test_error = (len(X_test) - np.sum(P_test == labels_test))/len(X_test)

# Print output
print('Training error: ' + str(train_error*100) + '%')
print('Testing error: ' + str(test_error*100) + '%')
```

B.2a

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from mnist import MNIST
from scipy import linalg

sns.set()

def train(X, Y, lam):
    d = int(X.size/len(X))
    reg_matrix = lam * np.eye(d)
    a = np.matmul(X.transpose(), X) + reg_matrix
    b = np.matmul(X.transpose(), Y)
    W = linalg.solve(a, b)
    return W

def predict(W, X):
    d = len(X)
    Y = np.zeros(d)
    temp = np.matmul(X, W)
    for i in range(0, d):
        ind = np.where(temp[i, :] == np.amax(temp[i, :]))
        Y[i] = ind[0]
    return Y

def hx(X, G, b):
    return np.cos(np.matmul(X, np.transpose(G)) + np.transpose(b))

# Load data
mndata = MNIST('./data/')
X_train, labels_train = map(np.array, mndata.load_training())
X_test, labels_test = map(np.array, mndata.load_testing())

# Rearrange data
X_train = X_train/255.0
X_test = X_test/255.0
Y_train = np.zeros((len(X_train), 10))
Y_train[range(0, len(X_train)), labels_train] = 1

# Split randomly into train and validation
inds = np.random.permutation(len(X_train))
samples = int(len(X_train)*0.8)
newX_train = X_train[inds[0:samples], :]
```

```
newX_test = X_train[inds[samples:len(inds)], :]
newY_train = Y_train[inds[0:samples], :]
newlabels_train = labels_train[inds[0:samples]]
newlabels_test = labels_train[inds[samples:len(inds)]]

# Cross-validation
mu = 0
var = 0.1
unilim = 2*np.pi
lam = 1e-4
d = int(np.size(newX_train)/ samples)
P = np.arange(50, 6000, 50)
train_error = np.zeros(len(P))
test_error = np.zeros(len(P))

for i in range(0, len(P)):
    p = P[i]
    print(str(p))
    G = np.random.normal(mu, np.sqrt(var), (p, d))
    b = np.random.uniform(0, unilim, (p, 1))

    # Apply function and train model
    h = hx(newX_train, G, b)
    W = train(h, newY_train, lam)

    # Training set output
    P_train = predict(W, h)

    # Apply function to validation and validate
    h = hx(newX_test, G, b)
    P_test = predict(W, h)

    # Calculate error
    train_error[i] = (len(P_train) - np.sum(P_train == newlabels_train))/len(P_train)
    test_error[i] = (len(P_test) - np.sum(P_test == newlabels_test)) / len(P_test)

# Plot
plt.plot(P, train_error*100)
plt.plot(P, test_error*100)
plt.xlabel('p')
plt.ylabel('% Error')
plt.legend(['Training Error', 'Validation Error'])
plt.show()
```