Homework 0 A

Fall 2020, CSE 546: Machine Learning
John Franklin Crenshaw
October 5, 2020

Collaborators: Samantha Tetef, Robert Pecoraro

Short Answer and "True or False" Conceptual Questions

A.0

- *Bias* is the difference between the expectation value of your model and the true value you want to learn. It is indicative of an error in your model assumptions, and won't disappear with more training data. In physics, we refer to this as a systematic error. *Variance* refers to the scatter in model predictions. Models with high variance are overfit, and thus are highly sensitive to changes in the training set. Training the model on different training sets, even when sampled from the same distribution, will result in very different predictions.
- When the complexity increases, bias decreases but variance increases. This is because a more complex model is able to capture the "real" information in the training set, and thus, on average, does well. However, the high complexity also allows the model to overfit and learn the random noise in the data, resulting in a model that is highly sensitive to changes in the training set. The reverse occurs as you decrease complexity.
- False. Bias is indicative of an incorrect model, and does not increase or decrease with the size of the training set.
- True. As you increase the size of the training set for a fixed-complexity model, the model is forced to generalize to more and more data points with the same number of parameters. This reduces the amount that the model overfits to the noise of the individual data points, and thus reduces the variance.
- False. The number of features contributes to model complexity. If your model is overfit, then decreasing complexity can improve generalization. However, if your model is underfit, then increasing complexity typically improves generalization. Usually the best generalization is at a happy medium of model complexity.
- You should use the training set. If you use the test set during the model-tweaking process, it isn't a test set anymore! It is now part of the training set. You should never use the test set for anything until you want to estimate how well your *final* model generalizes.
- False. As the training set is trained on the training set, the training set will give an over-optimistic estimate of the true error. This is what the test set is for - test the error of the model on a data set it has never seen before.

Maximum Likelihood Estimation (MLE)

A.1

a. The log likelihood is

$$\ell\left(\{x_i\}_{i=1}^N\right) = \log\left(\prod_{i=1}^N e^{-\lambda}\frac{\lambda^{x_i}}{x_i!}\right) = \sum_{i=1}^N(-\lambda + x_i\log\lambda - \log(x_i!)).$$

We can maximize this with respect to λ by taking the derivative and setting it equal to zero:

$$\frac{\partial\ell}{\partial\lambda} = \sum_{i=1}^N\left(-1 + \frac{x_i}{\lambda}\right) = 0 \quad\implies\quad \hat{\lambda} = \frac{1}{N}\sum_{i=1}^N$$

(Note we know this is a maximum as the second derivative is strictly negative). For our data, $\hat{\lambda} = 1.2$.

b. We can use the same general formula we derived above. That is, the MLE estimator is simply the mean of the goals scored, $\hat{\lambda} = 5/3 \approx 1.67$.

c. These are given in the answers to a and b. They are $\hat{\lambda} = 1.2$ and $\hat{\lambda} \approx 1.67$ respectively.

A.2 The PDF for this distribution is

$$p(x) = \begin{cases} \frac{1}{\theta} & 0 \leq x \leq \theta \\ 0 & \text{else.} \end{cases}$$

The likelihood is

$$\mathcal{L}\left(\{x_i\}_{i=1}^N\right) = \theta^{-N}.$$

We can obviously increase the likelihood by decreasing θ, until $\theta = \max\{x_i\}_{i=1}^N$, at which point decreasing θ makes the likelihood zero, as $p(x|x > \theta) = 0$. Thus $\hat{\theta} = \max\{x_i\}_{i=1}^N$.

Overfitting

A.3

a. As the train and test sets are both drawn from $\mathcal{D}$,

$$\mathbb{E}_{\text{train}}\left[(f(x) - y)^2\right] = \mathbb{E}_{\text{test}}\left[(f(x) - y)^2\right] = \mathbb{E}_{(x,y)\sim\mathcal{D}}\left[(f(x) - y)^2\right] = \epsilon(f).$$

Using this, we have

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

and the same for $\mathbb{E}_{\text{test}}[\hat{\epsilon}_{\text{test}}(f)]$:

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

Finally, we have

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

where in the last step, I used the fact that $\hat{f}$ does not change when averaging over test sets, as $\hat{f}$ depends only on the training set. Thus in this context, $\hat{f}$ is no different than f.

b. No this is not the case, as $\hat{f}$ changes with each instantiation of the training set, so we cannot substitute it for the general f in the definition of the true error. This is why the true error needs to be estimated on the test set.

c. Starting from the hint, we have

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

where the first equality on the second line uses the result of part a. The sum on the second line is averaging the expectation over all the functions $f \in \mathcal{F}$, which usually have greater uncertainty than the optimal f, so this average is greater than or equal to the error for the optimal f.

Bias-Variance Tradeoff

B.1

a. The complexity scales inversely with m, so I expect small m to have low bias, high variance and I expect large m to have the opposite.

b. Let's expand $\mathbb{E}[\hat{f}_m(x_i)]$:

$$\begin{aligned}
\mathbb{E}[\hat{f}_m(x_i)] &= \mathbb{E}\left[\sum_{j=1}^{n/m} c_j \, \mathbf{1}\{x_i \in \left(\frac{(j-1)m}{n}, \frac{jm}{n}\right]\}\right] \\
&= \frac{1}{m}\sum_{j=1}^{n/m} \sum_{k=(j-1)m+1}^{jm} \mathbb{E}[y_k] \, \mathbf{1}\{x_i \in \left(\frac{(j-1)m}{n}, \frac{jm}{n}\right]\} \\
&= \frac{1}{m}\sum_{j=1}^{n/m} \sum_{k=(j-1)m+1}^{jm} f(x_k) \, \mathbf{1}\{x_i \in \left(\frac{(j-1)m}{n}, \frac{jm}{n}\right]\} \\
&= \sum_{j=1}^{n/m} \bar{f}^{(j)} \, \mathbf{1}\{x_i \in \left(\frac{(j-1)m}{n}, \frac{jm}{n}\right]\} = \bar{f}^{(j \ni x_i)},
\end{aligned}$$

where I used the provided definitions for $\hat{f}_m(x_i)$, c_j, y_k, and $\bar{f}^{(j)}$, respectively, and in the final line I define $j \ni x_i$ as "the bin j to which the point x_i belongs." Now, we have

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

Instead of directly summing over every point x_i, we can instead sum over every bin j while summing over all the points in each bin. Redefining the index i to count over the points in a given bin j, we have

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

which is the desired result.

c. Plugging in the result for $\mathbb{E}[\hat{f}_m(x_i)]$ above,

$$\begin{aligned}
\mathbb{E}\left[\frac{1}{n}\sum_{i=1}^{n}(\hat{f}_m(x_i) - \mathbb{E}[\hat{f}_m(x_i)])^2\right] &= \mathbb{E}\left[\frac{1}{n}\sum_{i=1}^{n}(\hat{f}_m(x_i) - \bar{f}^{(j \ni x_i)})^2\right] \\
&= \mathbb{E}\left[\frac{1}{n}\sum_{i=1}^{n}(c_{j \ni x_i} - \bar{f}^{(j \ni x_i)})^2\right] \\
&= \mathbb{E}\left[\frac{1}{n}\sum_{j=1}^{n/m}\sum_{i=(j-1)m+1}^{jm}(c_j - \bar{f}^{(j)})^2\right] \\
&= \mathbb{E}\left[\frac{1}{n}\sum_{j=1}^{n/m} m(c_j - \bar{f}^{(j)})^2\right] = \frac{1}{n}\sum_{j=1}^{n/m} m\,\mathbb{E}\left[(c_j - \bar{f}^{(j)})^2\right].
\end{aligned}$$

In the second line, I plugged in the definition of $\hat{f}_m(x_i)$, again using the compact notation $j \ni x_i$ to denote the bin j to which the point x_i belongs. In the third line, I again converted the direct sum over x_i to a sum over bins j while summing over the points i in each bin. In the fourth line, I use the fact that the summand is independent of i, then apply linearity of expectation. This proves the first equality.

Now, using the definitions of c_j and $\bar{f}^{(j)}$, we have

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

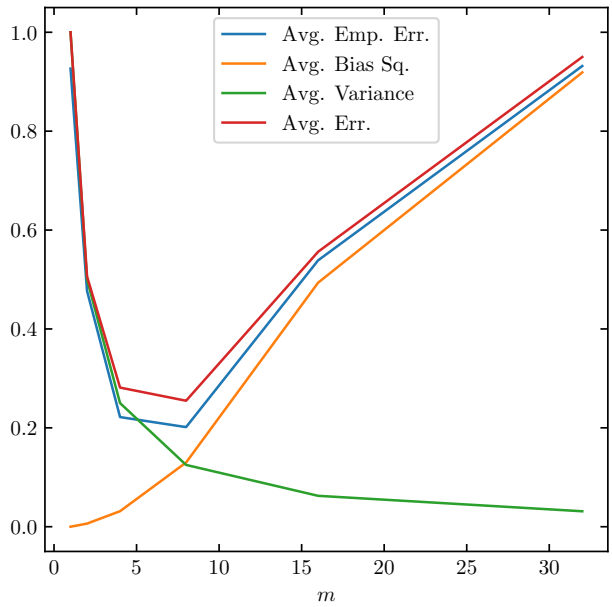
where in the last line, I used $\text{Var}(\epsilon) = \sigma^2 = \mathbb{E}[\epsilon^2] - \mathbb{E}[\epsilon]^2 = \mathbb{E}[\epsilon^2]$, and $E[\text{cross terms}] = 0$, as the noises ϵ_i are uncorrelated. Plugging this in above, we have

$$\begin{aligned}
\frac{1}{n}\sum_{j=1}^{n/m} m\,\mathbb{E}\left[(c_j - \bar{f}^{(j)})^2\right] &= \frac{1}{n}\sum_{j=1}^{n/m} m \cdot \frac{1}{m^2}\sum_{i=(j-1)m+1}^{jm}\sigma^2 \\
&= \frac{1}{mn}\left(\sum_{j=1}^{n/m}\sum_{i=(j-1)m+1}^{jm}\sigma^2\right) = \frac{\sigma^2}{m}.
\end{aligned}$$

Thus we have the final result,

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

d.



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

# set random seed
np.random.seed(11)
# create m and x grids
M = np.array([1, 2, 4, 8, 16, 32])
x = np.linspace(1/256, 1, 256)

# f and its estimator, fbar
f = 4 * np.sin(np.pi * x) * np.cos(6*np.pi * x**2)
fbar = lambda m: f.reshape(-1,m).mean(axis=1).repeat(m)
# y and its estimator, yhat
y = np.random.normal(f, 1)
fhat = lambda m: y.reshape(-1,m).mean(axis=1).repeat(m)

# the four quantities to plot
avg_emp_err = lambda m: np.mean( (fhat(m) - f)**2 )
avg_bias_sq = lambda m: np.mean( (fbar(m) - f)**2 )
avg_var = lambda m: 1/m
avg_err = lambda m: avg_bias_sq(m) + avg_var(m)

# make the plot
fig = plt.figure(figsize=(4,4), constrained_layout=True)
plt.plot(M, [avg_emp_err(m) for m in M], label='Avg. Emp. Err.')
plt.plot(M, [avg_bias_sq(m) for m in M], label='Avg. Bias Sq.')
plt.plot(M, [avg_var(m) for m in M], label='Avg. Variance')
plt.plot(M, [avg_err(m) for m in M], label='Avg. Err.')
plt.legend()
plt.xlabel('$m$')
```

e. The L-Lipschitz condition tells us that the maximum absolute value of the slope is L. The square difference $(\bar{f}^{(j)} - f(x_i))^2$ for each bin is strictly less than the square difference $(\max_{x_i \in j} f(x_i) - \min_{x_i \in j} f(x_i))^2$, which is less than or equal to $(L \cdot m/n)^2$, which is the difference assuming the function is monotonic with the maximum slope across the bin. Thus, the average biased squared is $\mathcal{O}(\frac{L^2 m^2}{n^2})$.

Using the formula for variance above, total error is $\mathcal{O}(\frac{L^2 m^2}{n^2} + \frac{\sigma^2}{m})$. Taking the derivative with respect to m and setting equal to zero:

$$\frac{2L^2 m}{n^2} - \frac{\sigma^2}{m^2} = 0 \implies m^3 = \frac{1}{2}\frac{n^2\sigma^2}{L^2}.$$

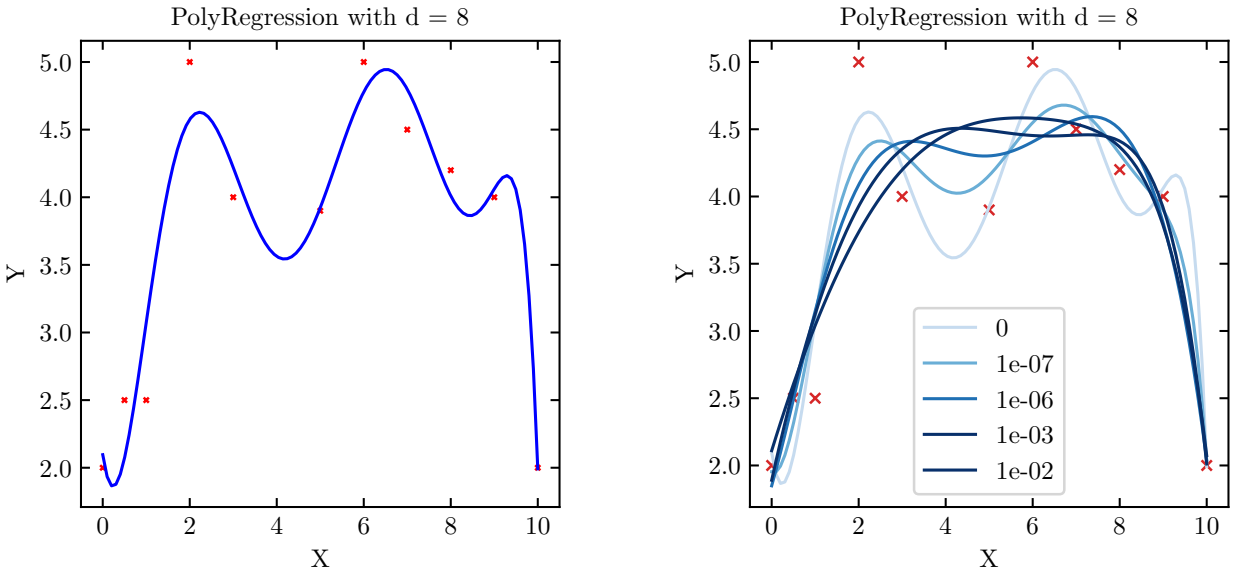
This scales as expected. To see this, remember that complexity scales inversely with m. Thus if L is large or σ^2 is small, we want a more complex model, which is expected for a quickly-changing variable with little noise. Furthermore, as you get more data points, you can afford to include more points in each bin, so m goes up with n, but not quite as fast: as you get more data points, the resolution of the bins can increase.

Plugging this value of m back into the error, we find that the error scales like $\mathcal{O}((\frac{L\sigma^2}{n})^{2/3})$. This is also as we expect: if the variable changes quickly and has high noise, we expect more error. We also expect less error as we increase the number of data points.

Polynomial Regression

A.4

Here are the results of using my PolynomialRegression implementation. On the left is the output of `test_polyreg_univariate.py`. On the right is the same, except with a range of different regularizations that are listed in the legend. You can see that increasing the regularization decreases the amplitude of the "wiggles" in the best fit curve. In the limit $\lambda \to \infty$, the best fit is the horizontal line $y = c$ where c is the offset that is fit regardless of λ.



Here is the code from `polyreg.py`:

```
class PolynomialRegression:

        def __init__(self, degree=1, reg_lambda=1E-8):
                """
                Constructor
                """
                self.degree = degree
                self.regLambda = reg_lambda
                self.theta = None
                self.Xmean = None
                self.Xstd = None

        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
```

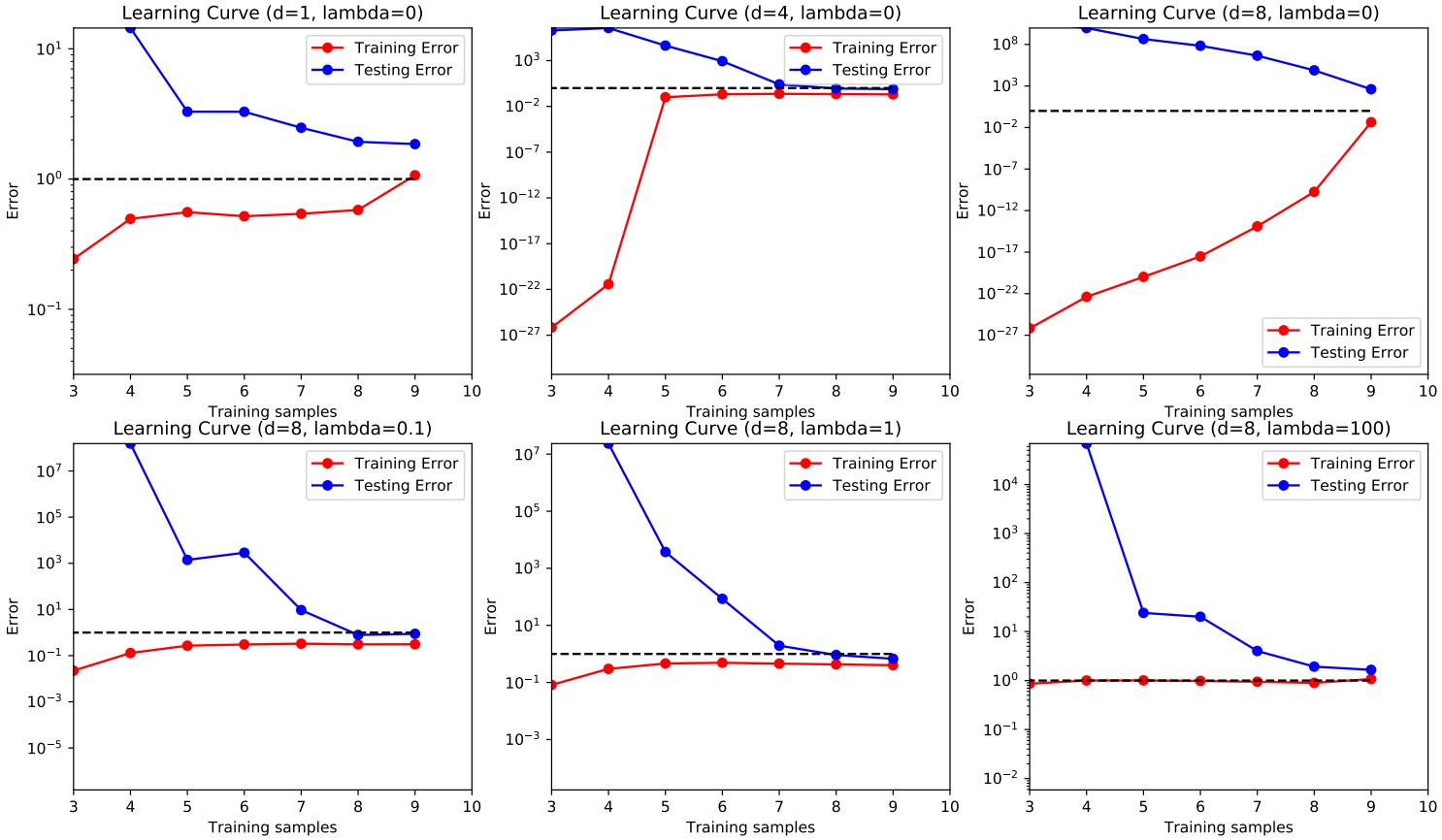
```
            """
            X_ = X.flatten()
            degree_ = np.arange(1, degree+1) # array of polynomial exponents 1 -> degree
            return np.power.outer(X_, degree_) # return polynomial features

    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
            """
            n = len(X)

            # matrix of polynomial features
            X_ = self.polyfeatures(X, self.degree)
            # standardize features
            self.Xmean = X_.mean(axis=0)
            self.Xstd = X_.std(axis=0)
            X_ = (X_ - self.Xmean) / self.Xstd
            # add column of ones to front, for x^0
            X_ = np.c_[np.ones(n), X_]

            # regularization matrix
            regMatrix = self.regLambda * np.identity(self.degree + 1)
            regMatrix[0, 0] = 0

            # analytic solution (X'X + regMatrix)^-1 X' y
            self.theta = np.linalg.pinv(X_.T.dot(X_) + regMatrix).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
            """
            n = len(X)

            # matrix of polynomial features
            X_ = self.polyfeatures(X, self.degree)
            # standardize features
            X_ = (X_ - self.Xmean) / self.Xstd
            # add column of ones to front, for x^0
            X_ = np.c_[np.ones(n), X_]

            return X_ @ self.theta
```

Here is the code to make the plot with many different values of λ:

```
# load the data
filePath = "data/polydata.dat"
file = open(filePath,'r')
allData = np.loadtxt(file, delimiter=',')

X = allData[:, [0]]
y = allData[:, [1]]

# plot curve
fig = plt.figure()
plt.plot(X, y, 'C3x', markersize=4)

# regression with degree = d
d = 8

# different lambdas to plots
lambdas = [0, 1e-7, 1e-6, 1e-3, 1e-2]
labels = ['0', '1e-07', '1e-06', '1e-03', '1e-02']

# colors for different lambdas
c = np.arange(len(lambdas))
norm = mpl.colors.Normalize(vmin=c.min(), vmax=c.max())
cmap = mpl.cm.ScalarMappable(norm=norm, cmap=mpl.cm.Blues)
cmap.set_array([])

for i in range(len(lambdas)):
    model = PolynomialRegression(degree=d, reg_lambda=lambdas[i])
    model.fit(X, y)
    xpoints = np.linspace(np.max(X), np.min(X), 100).reshape(-1, 1)
    ypoints = model.predict(xpoints)
    plt.plot(xpoints, ypoints, label=labels[i], c=cmap.to_rgba(i + 1))


plt.legend()
plt.title('PolyRegression with d = '+str(d))
plt.xlabel('X')
plt.ylabel('Y')
plt.show()

fig.savefig('../A4.2.pdf')
```

A.5

Here are the training curves. The results match the plot in the problem statement.




```
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 -- errorTrain[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)

    for i in range(1,n):

        # fit on the first i training points
        model = PolynomialRegression(degree=degree, reg_lambda=reg_lambda)
        model.fit(Xtrain[:i+1], Ytrain[:i+1])

        # predict on first i training points and compute error
        Ytrain_pred = model.predict(Xtrain[:i+1])
        errorTrain[i] = np.mean((Ytrain[:i+1] - Ytrain_pred)**2)

        # predict on whole test set and compute error
        Ytest_pred = model.predict(Xtest)
        errorTest[i] = np.mean((Ytest - Ytest_pred)**2)

    return errorTrain, errorTest
```

Ridge Regression on MNIST

A.6

a. We can write the given equation as

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

Now taking the derivative with respect to vector w_j and setting equal to zero, we have

$$\frac{\partial}{\partial w_j}[\cdots] = \sum_{j=0}^{k}\left[\hat{w}_j^T X^T X + \hat{w}_j^T X^T X - y_j^T X - y_j^T X + \lambda\hat{w}_j^T + \lambda\hat{w}_j^T\right]$$
$$= 2\sum_{j=0}^{k}\left[\hat{w}_j^T(X^T X + \lambda I) - y_j^T X\right] = 2\sum_{j=0}^{k}\left[(X^T X + \lambda I)\hat{w}_j - X^T y_j\right]^T = 0.$$

We can now see that

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

and as w_j and y_j are the columns of W and Y, respectively, we have

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

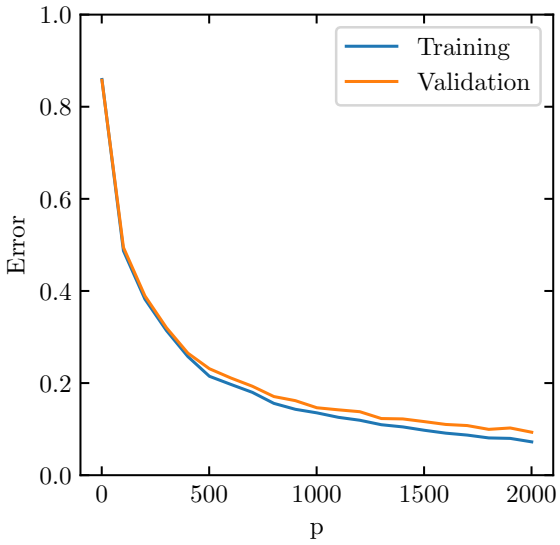
b. I get $\hat{\epsilon}_{\text{train}} = 0.1423$ and $\hat{\epsilon}_{\text{test}} = 0.1397$ with the following code:

```
import numpy as np
from mnist import MNIST

def load_dataset():
        mndata = MNIST('../../../python-mnist/data/')
        X_train, labels_train = map(np.array, mndata.load_training())
        X_test, labels_test = map(np.array, mndata.load_testing())
        X_train = X_train/255.0
        X_test = X_test/255.0

        return X_train, labels_train, X_test, labels_test

X_train, Y_train, X_test, Y_test = load_dataset()

class MNIST_Classifier():

        def __init__(self):
                self.W = None

        def train(self, X, Y, regLambda=0):

                n,d = X.shape

                # X matrix with ones column
                X_ = np.c_[np.ones(n), X]
```

```
                # regularization matrix
                regMatrix = regLambda * np.identity(d + 1)
                regMatrix[0, 0] = 0

                # Y matrix with one-hot encoding
                k = 10
                Y_ = np.identity(k)[Y]

                # analytical solution (X'X + regMatrix)^-1 X' Y
                self.W = np.linalg.solve(X_.T.dot(X_) + regMatrix, X_.T.dot(Y_))

        def predict(self, X):

                n,d = X.shape

                # X matrix with ones column
                X_ = np.c_[np.ones(n), X]

                # predict vectors
                Y = self.W.T.dot(X_.T).T

                # return argmax for each vector
                return Y.argmax(axis=1)

classifier = MNIST_Classifier()
classifier.train(X_train, Y_train, 1e-4)

diff_train = Y_train - classifier.predict(X_train)
e_train = len(diff_train[diff_train != 0]) / len(diff_train)
print(f'Train error = {e_train:.4f}')

diff_test = Y_test - classifier.predict(X_test)
e_test = len(diff_test[diff_test != 0]) / len(diff_test)
print(f'Test error = {e_test:.4f}')
```

B.2

a.



This plot was produced by the following code (which was run after the code listed in A.6.b):

```
np.random.seed(0)
# shuffle and split training set
idx_split = int(0.8 * len(X_train)) # where to split the set
idx_shuffle = np.random.permutation(np.arange(len(X_train))) # how to shuffle the set
X_train_new, X_val = X_train[idx_shuffle][:idx_split], X_train[idx_shuffle][idx_split:]
Y_train_new, Y_val = Y_train[idx_shuffle][:idx_split], Y_train[idx_shuffle][idx_split:]

# arrays to store errors
train_errs = []
val_errs = []

# loop over p
for p in np.arange(1,2002,100):

    # generate G and b
    G = np.random.normal(0, np.sqrt(0.1), (p,784))
    b = 2*np.pi * np.random.uniform(size=p)

    # calculate H
    H_train = np.cos((G @ X_train_new.T).T + b)
    H_val = np.cos((G @ X_val.T).T + b)

    # create a new classifier and train it
    classifier = MNIST_Classifier()
    classifier.train(H_train, Y_train_new, 1e-4)

    # calculate errors
    diff_train = Y_train_new - classifier.predict(H_train)
    e_train = len(diff_train[diff_train != 0]) / len(diff_train)
    train_errs.append(e_train)

    diff_val = Y_val - classifier.predict(H_val)
```

```
e_val = len(diff_val[diff_val != 0]) / len(diff_val)
val_errs.append(e_val)

# print every 200 steps
if p % 200 == 1:
    print(p)
```

b. We can calculate the confidence interval by using the equalities in the given equation, and setting $\delta = 0.05$. As the error can be anywhere in the range $[0, 1]$, we have $a = 0,\ b = 1$, and m is the number of samples in the validation set, which is $12,000$. Plugging these values in, we get

$$\sqrt{\frac{(1-0)^2 \log(2/0.05)}{2 \cdot 12000}} \approx 0.012,$$

assuming log is the natural logarithm. The best value of p I tested was $p = 2001$, for which the validation error was 0.093. So we have

$$\hat{\epsilon}_{\text{test}}(\hat{f}) = 0.093 \pm 0.012.$$

Note that from the graph, it appears likely that with a better computer, you could go to higher values of p and get even lower errors.