CSE546 HW1

Joseph David
(Worked with Kevin Liu)

April 21 2021

Short Answer and "True or False" Conceptual Questions

Problem A.0.

(a) Bias is the expected error of your model from the true expected value of the distribution as you fit your model to samples from the distribution. On the other hand, variance measures how much your model is expected to change when fitted to different samples. Bias-variance tradeoff is the idea that low model complexity may lead to high bias and low variance, but by increasing your model complexity, you may decrease bias while increasing the variance.

(b) This was covered in (a). Essentially, model complexity is negatively correlated with bias and positively correlated with variance.

(c) False, as more training data becomes available the expected value of the model should approach the true value and hence bias should decrease.

(d) True, the more data points there are reduces the impact that any single one has on the model.

(e) False, this can be true but is not always the case. (f) The training set, because then we know what the correct results should be. This method of using some of the training data to tune parameters is known as cross-validation.

(g) False, the training error provides an underestimate of the true error because the model is fitted to the training data and hence will be lower than the true error.

Maximum Likelihood Estimation (MLE)

Problem A.1.

(a) Following the hint, we compute the log of the likelihood. Let $\mathbf{x}$ denote the data vector. Since the x_i are independent, we have that

$$\log P(\mathbf{x}|\lambda) = \log\left(\prod_{j=1}^{5} e^{-\lambda}\frac{\lambda^{x_j}}{x_j!}\right) = \sum_{j=1}^{5}\log(e^{-\lambda}\frac{\lambda^{x_j}}{x_j!}) = \sum_{j=1}^{5}[-\lambda + x_j\log(\lambda) - \log(x_j!)].$$

Therefore,

$$\frac{d}{d\lambda}\log P(\mathbf{x}|\lambda) = \sum_{j=1}^{5} -1 + \frac{x_j}{\lambda}.$$

Setting this equal to 0 and solving for λ, we obtain

$$\widehat{\lambda}_{MLE} = \frac{\sum_{j=1}^{5} x_j}{\sum_{j=1}^{5} 1} = \frac{\sum_{j=1}^{5} x_j}{5}.$$

(b) Doing the same process as in part (a), we obtain

$$\widehat{\lambda}_{MLE} = \frac{\sum_{j=1}^{6} x_j}{\sum_{j=1}^{6} 1} = \frac{\sum_{j=1}^{6} x_j}{6}.$$

In general, for n data points, we see that

$$\widehat{\lambda}_{MLE} = \frac{\sum_{j=1}^{n} x_j}{\sum_{j=1}^{n} 1} = \frac{\sum_{j=1}^{n} x_j}{n},$$

which is the average of our data points.

(c) For $n = 5$, using our above expression, we see that $\widehat{\lambda}_{MLE} = 13/5 = 2.6$. For $n = 6$, we get $\widehat{\lambda}_{MLE} = 16/6 \approx 2.67$.

Problem A.2.

Since the x_i are independent random variables and we assume that they are uniformly distributed on $[0.\theta]$, we have that

$$\mathbb{P}(x_1, \ldots, x_n|\theta) = \begin{cases} 0 & \theta < \max\{x_1, \ldots, x_n\} \\ \theta^{-n} & \theta \geq \max\{x_1, \ldots, x_n\} \end{cases}.$$

Therefore,

$$\frac{d}{d\theta}\mathbb{P}(x_1, \ldots, x_n|\theta) = \begin{cases} 0 & \theta < \max\{x_1, \ldots, x_n\} \\ -n\theta^{-n-1} & x > \max\{x_1, \ldots, x_n\} \end{cases}.$$

Setting this equal to 0 and solving, we see that $\widehat{\theta}_{MLE} = \max\{x_1, \ldots, x_n\}$.

Overfitting

Problem A.3.

(a) Let f be a fixed model. Since we make no distinction in how we select training data points versus test data points, the expectation in $\mathbb{E}_{train}$ and $\mathbb{E}_{test}$ is taken over all possible sets of samples. The same is true for $\mathbb{E}_{(x,y)\sim D}$ and therefore, it must be true that for an arbitrary function f,

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

(b) This is not true in regards to the training loss. In this case, we have

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

To illustrate this, let $\widehat{f}$ be a model best fitted to some training data S_{train}. By definition, the error $\widehat{\epsilon}_{train}(\widehat{f})$ is as small as possibly can be obtained fitting to S_{train} within a given class of functions. Thus, following the argument of (a), S_{train} will be considered as test data at some point when computing $\mathbb{E}_{test}[\widehat{\epsilon}_{test}(\widehat{f})]$, and in these cases, the error in those terms will be greater than the error of the model that is fitted to the data as training data. Since this is true for all training data sets S_{train}, it must be true that $\mathbb{E}_{train}[\widehat{\epsilon}_{train}(\widehat{f})] \leq \epsilon(\widehat{f})$.

(c) By the hint, we have that

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

However, by the independence of the training and test data sets, we know that

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

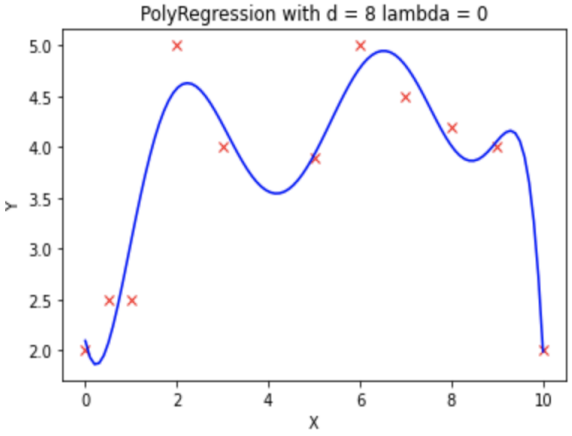
Furthermore, since $\widehat{\epsilon}_{train}(\widehat{f}_{train}) \leq \widehat{\epsilon}_{train}(f)$ for all $f \in \mathcal{F}$, we see that

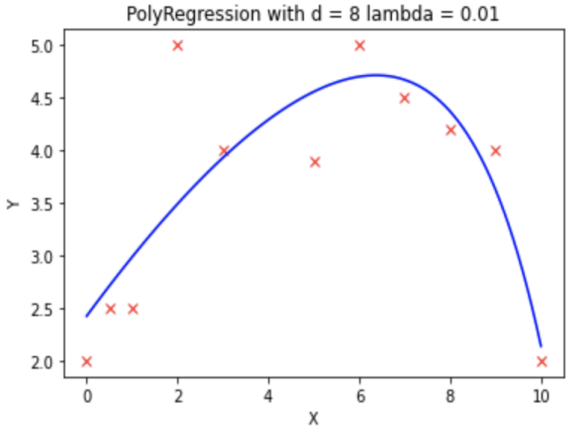
$$\begin{aligned}
\sum_{f\in\mathcal{F}} \mathbb{E}_{test}[\widehat{\epsilon}_{test}(f)]\mathbb{P}_{train}(\widehat{f}_{train} = f) &= \sum_{f\in\mathcal{F}} \mathbb{E}_{train}[\widehat{\epsilon}_{train}(f)]\mathbb{P}_{train}(\widehat{f}_{train} = f) \\
&\geq \sum_{f\in\mathcal{F}} \mathbb{E}_{train}[\widehat{\epsilon}_{train}(\widehat{f}_{train})]\mathbb{P}_{tr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

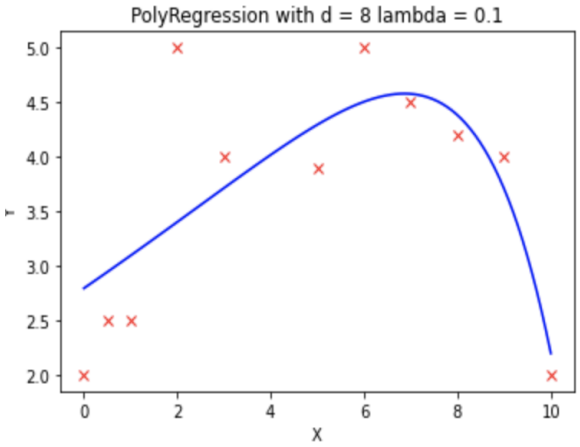
This proves the result.

Problem A.4.

The graphs below show the learned polynomial models of degree 8 and with varying regularization parameters λ. We see that as λ increases, the learned model seems to fit the data less and instead go through the center of the data points.







Below is the code defining the `PolynomialRegression` class.

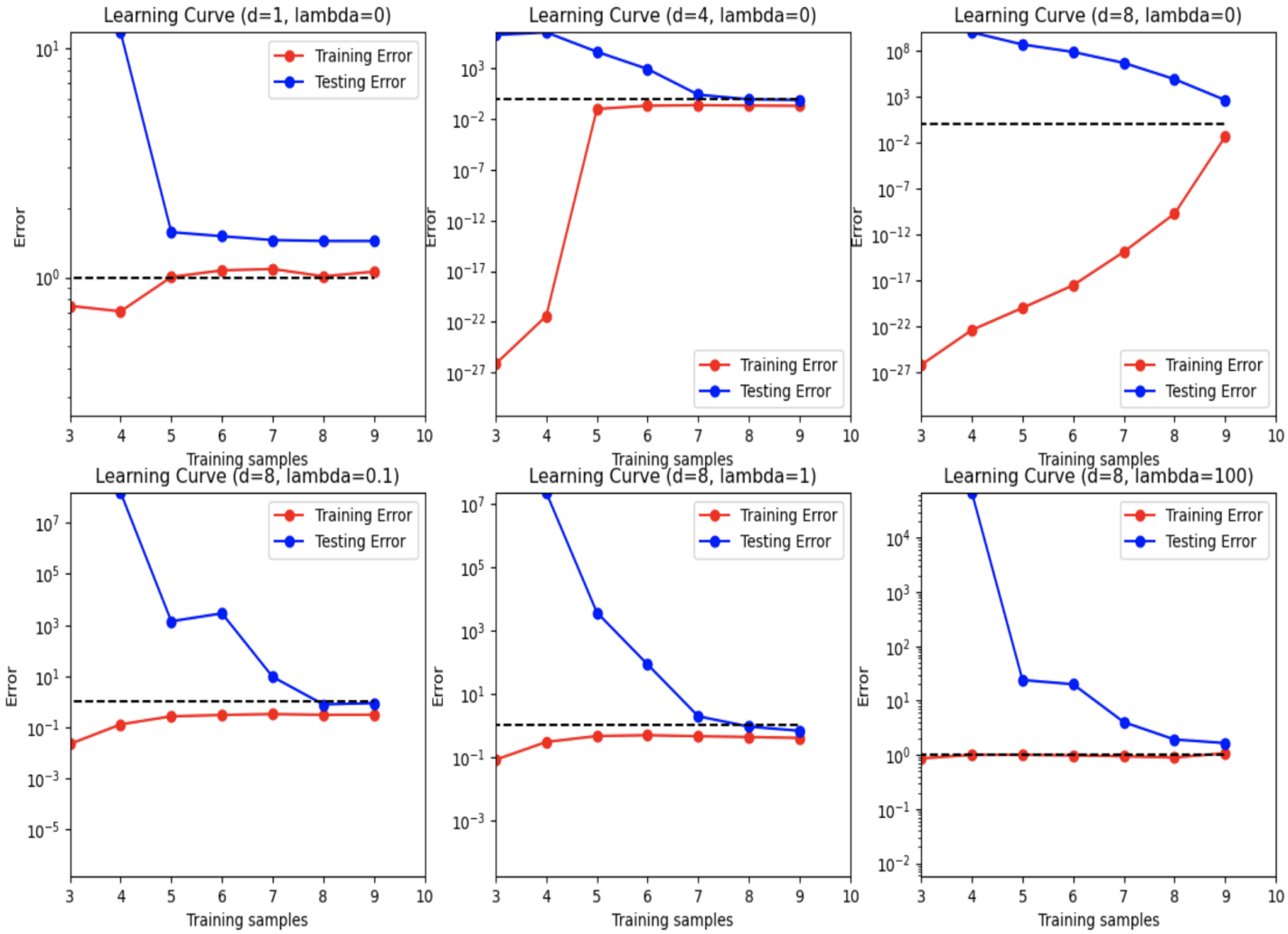
```
import numpy as np

class PolynomialRegression:

    def __init__(self, degree=1, reg_lambda=1E-8, means=0, std_dev=0 ):
        """
        Constructor
        """
        self.regLambda = reg_lambda
        self.degree = degree
        self.theta = None
        self.means = None
        self.std_dev = 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
        for j in range(2,degree+1):
            X_ = np.c_[X_, np.power(X,j)] #adds powers of X to the right

        return X_
```

```
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)
    d = self.degree

    #standardize data
    self.means = np.zeros([d,1])
    self.std_dev = np.zeros([d,1])

    X = self.polyfeatures(X,d)

    for j in range(d):
        self.means[j] = np.mean(X[:,j])
        self.std_dev[j] = np.std(X[:,j])
        if self.std_dev[j] == 0:
            self.std_dev[j] = 1
        for i in range(n):
            X[i,j] = (X[i,j]-self.means[j])/self.std_dev[j]

    X_ = np.c_[np.ones([n,1]),X]

    # construct reg matrix
    reg_matrix = self.regLambda * 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
    """
    n = len(X)
    d = self.degree

    #standardize data
    X = self.polyfeatures(X,d)
    for j in range(d):
        for i in range(n):
            X[i,j] = (X[i,j]-self.means[j])/self.std_dev[j]

    X = np.c_[np.ones([n,1]),X]

    return X.dot(self.theta)
```

Problem A.5.

Below are the figures showing training vs. testing error of polynomials of various degrees fitted using different values of regularization parameters. We see that as λ increases, the training error becomes worse, but on the other hand the testing error more quickly decreases with number of training samples. The top three graphs show the figures when $\lambda = 0$ but with increasing model complexity. From these graphs, we see that with more model complexity we may see very low training error but higher testing error. This is because we are overfitting our model to the training data and therefore does not generalize well to new data.



Below is the code for the function `learningCurve`, which was used along with class `PolynomialRegression` along with `testPolyregLearningCurve.py` to generate the above graphs.

```python
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)

    d = degree
    lam = reg_lambda

    for j in range(n):
        model = PolynomialRegression(degree=d, reg_lambda=lam)
        model.fit(Xtrain[0:j+1], Ytrain[0:j+1])
        prediction_train = model.predict(Xtrain[0:j+1])
        prediction_test = model.predict(Xtest)

        errorTrain[j] = np.sum(np.square(Ytrain[0:j+1] - prediction_train)) / len(Xtrain[0:j+1])
        errorTest[j] = np.sum(np.square(Ytest - prediction_test)) / len(Xtest)

    return errorTrain, errorTest
```

Problem A.6.

(a) Let $R(D,W) = ||W^TX - Y||^2 + \lambda||W||_F^2$, where $\lambda > 0$. Thus, we wish to find the $\widehat{W}$ such that $R(D,\widehat{W})$ is minimal. Expanding, we see that

$$\begin{aligned} R(D,W) &= ||W^TX - Y||^2 + \lambda||W||_F^2 \\ &= (W^TX - Y)^T(W^TX - Y) + \lambda||W||_F^2 \\ &= X^TWW^TX - Y^TW^TX - X^TWY + Y^TY + \lambda||W||_F^2. \end{aligned}$$

Therefore, differentiating $R(D,W)$ with respect to W, setting that equal to 0 and solving for $\widehat{W}$, we have that

$$\begin{aligned} 0 &= \frac{\partial R(D,W)}{\partial W} \\ &= 2X^TXW - Y^TX - X^TY + 2\lambda W \\ &= 2X^TXW - 2X^TX + 2\lambda W \\ &\Rightarrow 0 = X^TX\widehat{W} - X^TX + \lambda\widehat{W} \\ &\Rightarrow X^TX = (X^TX + \lambda I)\widehat{W} \\ &\Rightarrow \widehat{W} = (X^TX + \lambda I)^{-1}X^TY. \end{aligned}$$

Note that we have used the fact that, by definition of the Froebenius norm, $||W||_F^2 = \sum_{i=1}^{d}\sum_{j=1}^{k} W_{i,j}^2$, we have that

$$\frac{\partial||W||_F^2}{\partial W_{i,j}} = 2W_{i,j},$$

and therefore $\partial||W||_F^2/\partial W = 2W$.

(b)

```
def load_dataset():
    mndata = MNIST('/Users/joseph/Courses/cse546/hw1/mnist')
    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]
```

```
def train(X,Y,lam=0):
    if lam < 0:
        print("Lambda must be positive")
        return
    else:
        return np.matmul(np.matmul(np.linalg.inv(np.matmul(X.transpose(),X)
                                                 + lam*np.identity(len(X[1,:]))),X.transpose()),Y)
```

```
def predict(W,X2):
    m = len(X2[:,1])
    predictions = np.zeros(m)
    for j in range(m):
        x = np.matmul(W.transpose(),X2[j,:].transpose())
        predictions[j] = (np.argmax(x)).astype(int)
    return predictions
```

```
[X_train,labels_train,X_test,labels_test]=load_dataset()
```

```
n=len(X_train[:,1])
d=len(X_train[1,:])
k=10
m=len(X_test[:,1])
Y=np.zeros((n,k))
for j in range(n):
    Y[j,labels_train[j]] = 1
```

```
lam = 10**(-4)
What = train(X_train,Y,lam)
pred_train = predict(What,X_train)
pred_test = predict(What,X_test)
```

```
error_train=0
error_test=0

for j in range(n):
    if pred_train[j] != labels_train[j]:
        error_train = error_train + 1
for j in range(m):
    if pred_test[j] != labels_test[j]:
        error_test = error_test + 1

error_train = error_train / n
error_test = error_test / m
```

```
print(error_train)
print(error_test)
```

```
0.14805
0.1466
```

Bias-Variance Tradeoff

Problem B.1.

(a) For small values of m, we partition the x_i into smaller subsets and hence $\widehat{f}_m$ will attain many values (in particular it will attain n/m values). Thus, we would expect the bias to be small and the variance to be large for small values of m. On the other hand, for large values of m we have fewer partition groups and hence $\widehat{f}_m$ will attain fewer values and only attain the averages over larger subsets of the data. Thus, we expect to see higher bias and smaller variance for large values of m.

(b) First, note that if $x_i \in \left(\frac{(j-1)m}{n}, \frac{jm}{n}\right)$, then by definition $\widehat{f}_m$, we have that

$$\begin{aligned}
\mathbb{E}[\widehat{f}_m(x_i)] &= \mathbb{E}[c_j] \\
&= \mathbb{E}\left[\frac{1}{m}\sum_{i=(j-1)m+1}^{jm} y_i\right] \\
&= \mathbb{E}\left[\frac{1}{m}\sum_{i=(j-1)m+1}^{jm} f(x_i) + \epsilon_i\right] \\
&= \frac{1}{m}\left(\mathbb{E}\left[\sum_{i=(j-1)m+1}^{jm} f(x_i)\right] + \mathbb{E}\left[\sum_{i=(j-1)m+1}^{jm} \epsilon_i\right]\right) \\
&= \frac{1}{m}\left(\sum_{i=(j-1)m+1}^{jm} f(x_i) + 0\right) \\
&= \bar{f}^{(j)} + 0 = \bar{f}^{(j)}.
\end{aligned}$$

Therefore, we have that

$$\begin{aligned}
\frac{1}{n}\sum_{i=1}^{n}\left(\mathbb{E}[\widehat{f}_m(x_i)] - f(x_i)\right)^2 &= \frac{1}{n}\sum_{j=1}^{n/m}\sum_{i=(j-1)m+1}^{jm}\left(\mathbb{E}[\widehat{f}_m(x_i)] - f(x_i)\right)^2 \text{ (by change of indices)} \\
&= \frac{1}{n}\sum_{j=1}^{n/m}\sum_{i=(j-1)m+1}^{jm}\left(\bar{f}^{(j)} - f(x_i)\right)^2.
\end{aligned}$$

(c) We again use the fact that $\bar{f}^{(j)} = \mathbb{E}[\widehat{f}_m(x_i)]$ when $x_i \in ((j-1)m/n, jm/n]$. We see that

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

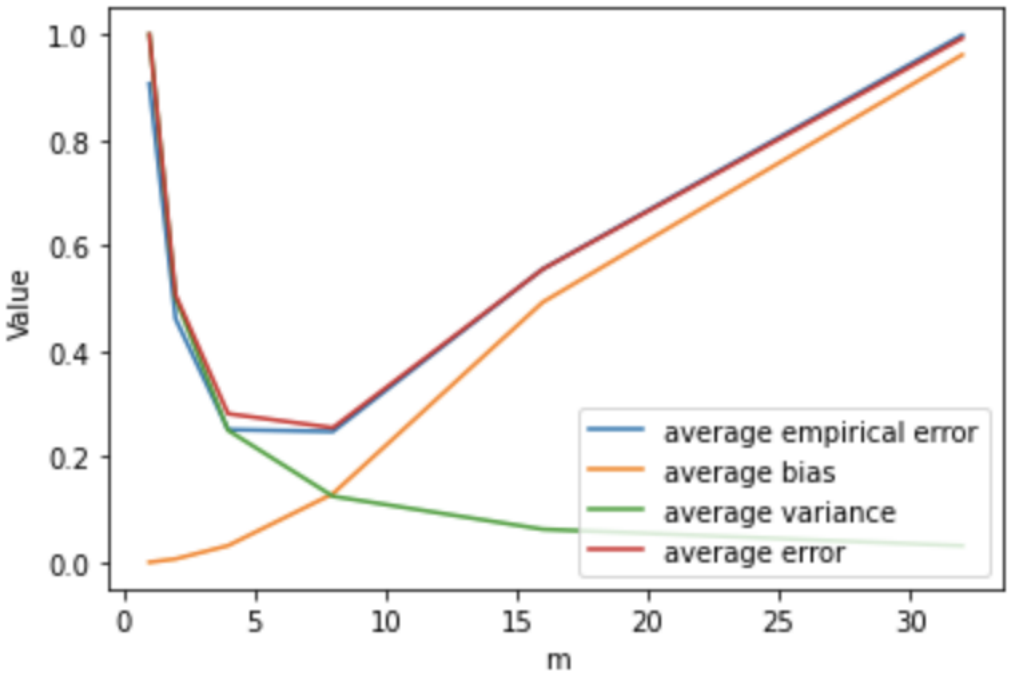
Next, notice that if we treat each c_j as a random variable then we have that, for each $j = 1, \ldots, n/m$, $\mathbb{E}[c_j] = \bar{f}^{(j)}$ and

hence $\mathbb{E}[(c_j - \bar{f}^{(j)})^2] = Var(c_j)$. However,

$$\begin{aligned}
Var(c_j) &= Var\left(\frac{1}{m}\sum_{i=(j-1)m+1}^{jm} f(x_i) + \epsilon_i\right) \\
&= Var\left(\frac{1}{m}\sum_{i=(j-1)m+1}^{jm} f(x_i)\right) + Var\left(\frac{1}{m}\sum_{i=(j-1)m+1}^{jm} \epsilon_i\right) \\
&= 0 + Var\left(\frac{1}{m}\sum_{i=(j-1)m+1}^{jm} \epsilon_i\right) \\
&= \frac{1}{m^2} Var\left(\sum_{i=(j-1)m+1}^{jm} \epsilon_i\right) \\
&= \frac{1}{m^2}\left(\sum_{i=(j-1)m+1}^{jm} Var(\epsilon_i) + \sum_{i\neq j} Cov(\epsilon_i, \epsilon_j)\right) \quad (Var(\epsilon_i) = \sigma^2 \text{ for all } i \text{ and } Cov(\epsilon_i, \epsilon_j) = 0 \text{ for } i \neq j) \\
&= \frac{1}{m^2} m\sigma^2 \\
&= \frac{\sigma^2}{m},
\end{aligned}$$

which proves the result.

(d) Below are the graphs for average empirical error, average bias, average variance, and average error for $m = 1, 2, 4, 8, 16, 32$, where we are using the above method of fitting a step function to the function $f(x) = 4\sin(\pi x)cos(\pi x^2)$ using $n = 256$ data points with Gaussian noise with $\sigma^2 = 1$. we see that our expectations from part (a) are mostly correct. The model's bias increases, and variance decreases, as m increases.



(e) Note that

$$\begin{aligned}
|c_j - \bar{f}^{(j)}|^2 &\leq |\min f(x_i) - \max f(x_i)|^2 \qquad (\text{since } \min f(x_i) \leq c_j, \bar{f}^{(j)} \leq \max f(x_i)) \\
&\leq \frac{L^2}{n^2}|i-j|^2 \qquad\qquad (\text{since } f \text{ is L-lipschitz}) \\
&\leq \frac{L^2}{n^2} m^2.
\end{aligned}$$

By part (b), this proves that average bias-squared is $O(\frac{L^2m^2}{n^2})$. Then, by part (c), we have that $O(\frac{L^2m^2}{n^2} + \frac{\sigma^2}{m})$. Taking the derivative of this expression with respect to m and setting this equal to 0, we obtain

$$0 = \frac{d}{dm}\left(\frac{L^2m^2}{n^2} + \frac{\sigma^2}{m}\right) = \frac{2L^2m}{n^2} - \frac{\sigma^2}{m^2}.$$

Solving for m, we see that

$$m = \left(\frac{\sigma^2 n^2}{2L^2}\right)^{2/3}.$$

Plugging this back in to our original expression, we see that the total error is on the order of

$$\left(\frac{L\sigma^2}{n}\right)^{2/3} + \left(\frac{\sigma L^2}{n^2}\right)^{2/3}.$$

Thus, we see that the total error decreases with large n, which seems plausible intuitively since this is the number of sample points we use. On the other hand, total error increases with large σ and L, which also seems plausible.
Below is the code.

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

```
n=256
sigma=1
M=[1,2,4,8,16,32]
x=np.zeros(n)
f=np.zeros(n)
fhat=np.zeros((len(M),n))
```

```
for j in range(n):
    x[j] = j/n
    f[j] = 4*np.sin(np.pi*x[j])*np.cos(6*np.pi*x[j]**2)
```

```
plt.plot(x,f)
```

...

```
error = np.random.normal(0,1,n)
y = f + error
plt.plot(x,y)
```

...

```
for k in range(len(M)):
    for j in range(int(n/M[k])):
        for i in range(j*M[k],(j+1)*M[k]):
                fhat[k,i] = (1/M[k])*np.sum(y[j*M[k]:(j+1)*M[k]])
```

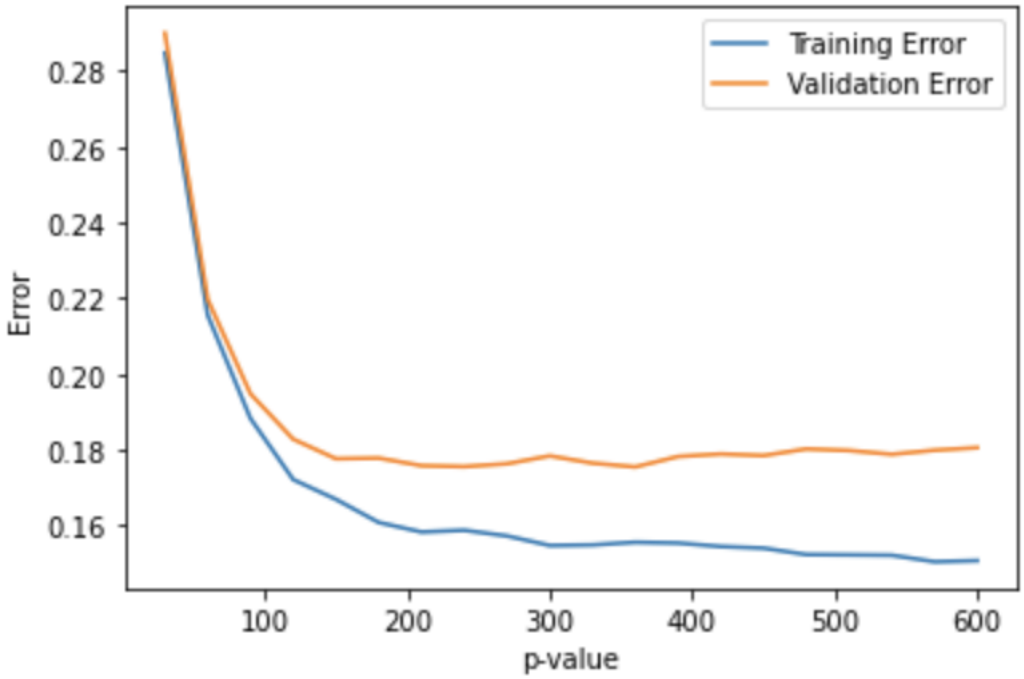
```
fbar = np.zeros((len(M),n))
for k in range(len(M)):
    for j in range(int(n/M[k])):
        for i in range(j*M[k],(j+1)*M[k]):
            fbar[k,i] = (1/M[k])*np.sum(f[j*M[k]:(j+1)*M[k]])
plt.plot(x,f)
plt.step(x,fbar[4,:])
```

...

```
avg_error=np.zeros(len(M))
avg_bias=np.zeros(len(M))
avg_var=np.zeros(len(M))
for k in range(len(M)):
    avg_error[k] = (1/n)*np.sum(np.square(fhat[k,:] - f))
    avg_bias[k] = (1/n)*np.sum(np.square(fbar[k,:] - f))
    avg_var[k] = 1/M[k]
plt.plot(M,avg_error, label="average empirical error")
plt.plot(M,avg_bias,label="average bias")
plt.plot(M,avg_var,label="average variance")
plt.plot(M, avg_bias+avg_var,label="average error")
plt.legend()
plt.xlabel("m")
plt.ylabel("Value")
```

Problem B.2.

(a) The figure shows the Training vs Validation Error for various values of p, where p is the dimension of the space that we randomly transformed our data into. We see that at around $p = 200$ the validation error seems to level out, indicating diminishing returns on increasing p.



Below is the code used to produce the above figure. The functions `loaddataset`, `train`, and `predict` from A.6 were used as well.

```
P=[(j+1)*30 for j in range(20)] #P=[100,200,...,3000]
d=784
n=60000
k=10
lam=10**(-4)
mu=0
sigma=np.sqrt(.1)

#create validation and training sets
perm = np.random.permutation(np.arange(n))
validation_data = X_train[perm[0:12000],:]
training_data = X_train[perm[12000:n],:]

#create labels matrix Y
Y=np.zeros((n,k))
for j in range(n):
    Y[j,labels_train[j]] = 1
Y_train=Y[perm[12000:n],:]
labels_val=labels_train[perm[0:12000]]
labels_train_sub=labels_train[perm[12000:n]]
```

```
errors_val = np.zeros(len(P))
errors_train = np.zeros(len(P))
counter=0
for p in P:
    G=np.random.normal(0,sigma,size=(p,d))
    b=np.random.uniform(0,2*np.pi,p)
    h_x_train = np.matmul(G,training_data.transpose())
    h_x_val = np.matmul(G,validation_data.transpose())
    for j in range(p):
        h_x_train[:,j] = np.cos(h_x_train[:,j] + b)
        h_x_val[:,j] = np.cos(h_x_val[:,j] + b)
    h_x_train = h_x_train.transpose()
    h_x_val = h_x_val.transpose()

    What = train(h_x_train,Y_train,lam)
    pred_train = predict(What,h_x_train)
    pred_val = predict(What,h_x_val)

    error_train=0
    error_val=0
    for j in range(48000):
        if pred_train[j] != labels_train_sub[j]:
            error_train = error_train + 1
    for j in range(12000):
        if pred_val[j] != labels_val[j]:
            error_val = error_val + 1

    errors_train[counter] = error_train / 48000
    errors_val[counter] = error_val / 12000

    counter = counter + 1
```

```
plt.plot(P,errors_train,label="Training Error")
plt.plot(P,errors_val,label="Validation Error")
plt.legend()
plt.xlabel("p-value")
plt.ylabel("Error")
```

(b) From the results in part (a), we see that the validation error seems to level out around $p = 250$, so we set $\widehat{p} = 250$. Then we computed the test error and found that $\widehat{\epsilon}_{test}(\widehat{f}) = 0.1689$. Since the data has $m = 10,000$ instances, we define X_i (for $i = 1, \ldots, 10,000$) to be random variables such that $X_i = 0$ if the ith data point is classified correctly and $X_i = 1$ if incorrectly classified. With these definitions, we see that

$$\frac{1}{m}\sum_{i=1}^{m} X_i = \widehat{\epsilon}_{test}(\widehat{f})$$

and $\mu = \mathbb{E}_{test}[\widehat{\epsilon}_{test}(\widehat{f})]$. Thus, with $a = 0$, $b = 1$, and $\delta = .05$, using Hoeffding's inequality we see that

$$\mathbb{P}\left(\left|\widehat{\epsilon}_{test}(\widehat{f}) - \mathbb{E}_{test}[\widehat{\epsilon}_{test}(\widehat{f})]\right| \geq \sqrt{\frac{\log(2/.05)}{2(10,000)}}\right) \leq .05.$$

Computing $\sqrt{\log(2/.05)/20,000} \approx .00895$ and since we computed that $\widehat{\epsilon}_{test}(\widehat{f}) \approx 0.1689$, we see that with 95% confidence

$$0.15995 \leq \mathbb{E}_{test}[\widehat{\epsilon}_{test}(\widehat{f})] \leq 0.17785.$$

Below is the code and result for $\widehat{\epsilon}_{test}(\widehat{f})$.

```
phat=250
G=np.random.normal(0,sigma,size=(phat,d))
b=np.random.uniform(0,2*np.pi,phat)
h_x=np.matmul(G,X_train.transpose())
h_x_test=np.matmul(G,X_test.transpose())
for j in range(phat):
    h_x[:,j] = np.cos(h_x[:,j]+b)
    h_x_test[:,j] = np.cos(h_x_test[:,j]+b)
h_x = h_x.transpose()
h_x_test = h_x_test.transpose()
What = train(h_x,Y,lam)
predictions = predict(What,h_x_test)
```

```
error_test_phat=0
for j in range(len(X_test[:,1])):
    if predictions[j] != labels_test[j]:
        error_test_phat = error_test_phat + 1
error_test_phat = error_test_phat / len(X_test[:,1])
error_test_phat
```

```
0.1689
```