Homework #1 - B

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

Bias-Variance tradeoff

B.1

a. *[5 points]* For small m bias should go down and variance up, as we reach higher complexity and closer to the specific data points. Conversely, for large m bias should go up and variance down 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 and all other values are 0. Therefore, we can remove the summation and the $\mathbf{1}$ function while keeping in mind that i must be between $(j-1)m+1$ and jm:

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

The second to last step is due to the constraint that i is between $(j-1)m+1$ and jm as all other values are 0. As a result, the equality 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:

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

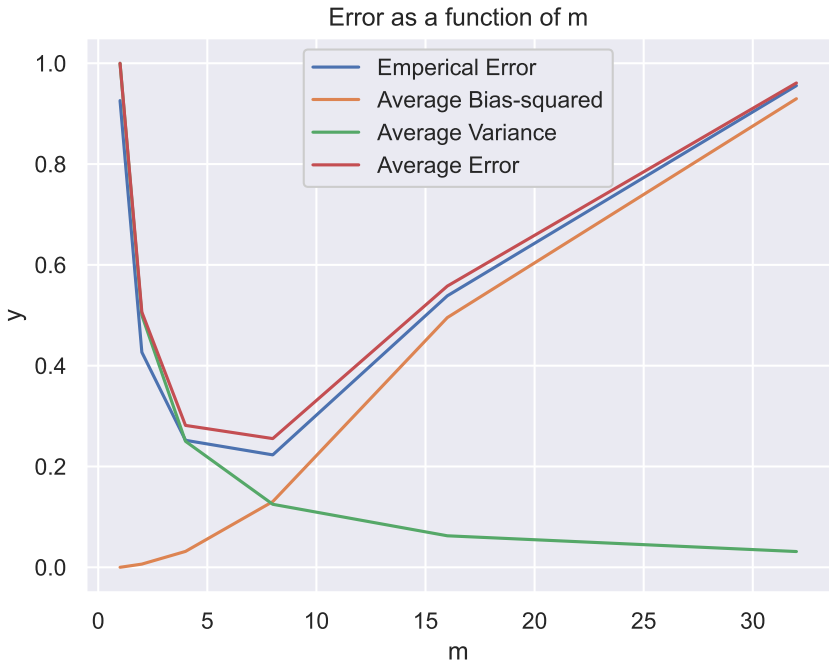
which resolves the first equality. Now we can look at the variance of c_j for the second equality:

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

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

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

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.



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()
```

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:

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

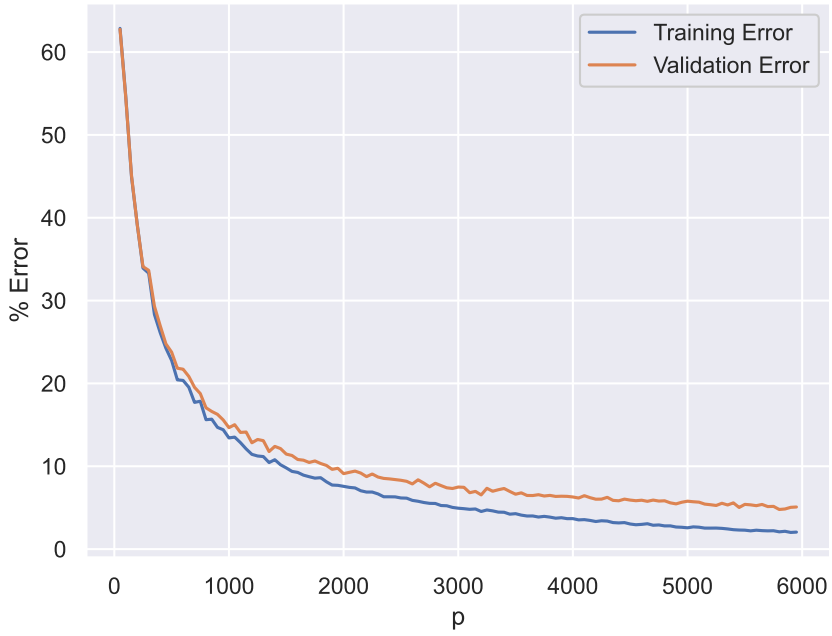
Plugging it back into the total error expression, we get:

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

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.

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.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()
```

b. *[5 points]* The value of p that gives the lowest error is 6000 as it continues to decrease. Since we are looking at error, we can define the following:

$$X_i = \mathbf{1}(\hat{f}(x_i) \neq z_i)$$
$$\frac{1}{m}\sum_{i=1}^{m} X_i = \hat{\epsilon}_{\text{test}}(\hat{f})$$
$$\mu = \mathbb{E}_{\text{test}}[\hat{\epsilon}_{\text{test}}(\hat{f})](\text{true error})$$
$$b = 1$$
$$a = 0$$
$$\delta = 0.05$$
$$m = \text{size of test set}$$

With these values, the test error with 95% confidence interval is $5.23\% \pm 1.3581\%$

B.2b

```
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], :]
newY_train = Y_train[inds[0:samples], :]

mu = 0
var = 0.1
unilim = 2*np.pi
lam = 1e-4
d = int(np.size(newX_train)/ samples)
p = 6000

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)

# Apply function to test set and validate
h = hx(X_test, G, b)
P_test = predict(W, h)

# Calculate error
test_error = (len(P_test) - np.sum(P_test == labels_test)) / len(P_test)

delta = 0.05
a = 0
b = 1
m = len(X_test)

conf_int = np.sqrt((b-a)**2*np.log(2/delta)/(2*m))

print(test_error)
print(conf_int)
```