CSE 446 HW1

Zachary McNulty

April 2019

1 Problem 1

1. (a) Let $X_1, X_2, ..., X_n \sim Poisson(\lambda)$ be a sequence of iid random variables. Then given a sequence of values $(x_1, x_2, ..., x_n)$ that these values take on the Maximum Likelihood Estimate of the parameter λ is:

$$P(x_1, x_2, ..., x_n \mid \lambda) = L_n(\lambda) = \prod_{i=1}^{n} P(X_i = x_i \mid \lambda)$$

$$log(L_n(\lambda)) = \sum_{i=1}^{n} log\left(P(X_i = x_i \mid \lambda)\right) = \sum_{i=1}^{n} log\left(\frac{e^{-\lambda}\lambda^{x_i}}{x_i!}\right)$$

$$= \sum_{i=1}^{n} \left(log(e^{-\lambda}) + log(\lambda^{x_i}) - log(x_i!)\right)$$

$$= \sum_{i=1}^{n} \left(-\lambda + x_i log(\lambda) - log(x_i!)\right)$$

$$\frac{\delta log(L_n(\lambda))}{\delta \lambda} = \sum_{i=1}^{n} \left(-1 + \frac{x_i}{\lambda}\right) = 0$$

$$\frac{n * mean(x_i)}{\lambda} = n$$

$$\lambda = mean(x_i)$$

(b)

$$\hat{\lambda} = \frac{1}{6}(1 + 3 + 3 + 0 + 1 + 5) = \frac{13}{6}$$

2 Problem 2

Let $X_1, X_2, ..., X_n \sim Unif[0, \theta]$ b a sequence of random variables. Suppose a sample of these random variables generates the corresponding non-negative values $x_1, ..., x_n$. As X_i are uniform on $[0, \theta]$ they have the following density function f :

$$f(x) = \begin{cases} \frac{1}{\theta} & x \in [0, \theta] \\ 0 & otherwise \end{cases}$$

Thus the MLE for θ is:

$$P(x_1, x_2, ..., x_n \mid \theta) = L_n(\theta) \sim \prod_{i=1}^{n} f(x_i)$$

Clearly, if $x_i > \theta$ for any i then the likelihood of that event occurring given θ is zero, making this choice of θ certainly suboptimal. Furthermore, as we know x_i come from some uniform distribution on $[0, \theta]$ we know they all must be non-negative. Thus, we can assume $0 \leq x_i \leq \theta \ \forall i$ for the optimal θ. This yields:

$$\prod_{i=1}^{n} f(x_i) = \prod_{i=1}^{n} \frac{1}{\theta} = \frac{1}{\theta^n}$$

This likelihood function is a decreasing value of θ so the optimal choice must be the lowest possible value of θ. Since we got the values $x_1, x_2, ..., x_n$ already we know these values must be possible to obtain and thus in $[0, \theta]$. Therefore, $\theta \geq max_i(x_i)$. Thus, the optimal value of θ is $\theta = \max_i(x_i)$ as this is as low as possible.

3 Problem 3

Problem 3a)

Let T a possible training set $[(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)]$ sampled from the overall distribution $\mathcal{D}$. Suppose $n = N_{train}$ so that our sample includes N_{train} points. Fix f to be function $f : x \to y$.

$$E_{train}[\hat{\epsilon}_{train}(f)] = \sum_{T \subset \mathcal{D}} P(T)\hat{\epsilon}_T(f)$$

Since f is fixed, it is the same regardless of which training dataset is used. Thus this expands to:

$$= \sum_{T \subset \mathcal{D}} P\left((x_1, y_1), (x_2, y_2), ..., (x_n, y_n)\right) \frac{1}{N_{train}} \sum_{(x,y) \in T} (y - f(x))^2$$

Assuming our data points $[(x_1, y_1), ..., (x_n, y_n)]$ are sampled independently from the distribution $\mathcal{D}$ we can separate the above probability:

$$= \frac{1}{N_{train}} \sum_{T \subset \mathcal{D}} P((x_1, y_1))P((x_2, y_2)) * ... * P(x_n, y_n) \sum_{(x,y) \in T} (y - f(x))^2$$

$$= \frac{1}{N_{train}} \sum_{(x_1,y_1) \in \mathcal{D}} \sum_{(x_2,y_2) \in \mathcal{D}} *...* \sum_{(x_n,y_n) \in \mathcal{D}} P((x_1, y_1))P((x_2, y_2))*...*P(x_n, y_n) \sum_{(x,y) \in T} (y-f(x))^2$$

From here, we can pull out terms from the inner sums as these variables are only looped over in outer sums:

$$= \frac{1}{N_{train}} \sum_{(x_1,y_1) \in \mathcal{D}} P((x_1, y_1)) \sum_{(x_2,y_2) \in \mathcal{D}} P((x_2, y_2))*...* \sum_{(x_n,y_n) \in \mathcal{D}} P((x_n, y_n)) \sum_{(x,y) \in T} (y-f(x))^2$$

Note that we can expand the final term as follows:

$$\sum_{(x,y) \in T} (y - f(x))^2 = (y_1 - f(x_1))^2 + (y_2 - f(x_2))^2 + ... + (y_n - f(x_n))^2$$

This allows us to separate the above sums into a series of terms by distributing them across these inner sums. This generates:

$$= \frac{1}{N_{train}} \sum_{i=1}^{n} \sum_{(x_1,y_1)\in\mathcal{D}} P((x_1,y_1)) * ... * \sum_{(x_n,y_n)\in\mathcal{D}} P((x_n,y_n))(y_i - f(x_i))^2$$

Since this inner term $(y_i - f(x_i))^2$ does not depend on some of the inner sums, we can pull it outwards.

$$= \frac{1}{N_T} \sum_{i=1}^{n} \sum_{(x_1,y_1)} P((x_1,y_1)) * ... * \sum_{(x_i,y_i)} P((x_i,y_i))(y_i - f(x_i))^2 * ... * \sum_{(x_n,y_n)} P((x_n,y_n))$$

Note that all the terms to the right of this $P(x_i, y_i)(y_i - f(x_i))^2$ term are just the sums over a probability distribution and thus naturally sum to one. Also note that this i^{th} term is exactly an expectation. In fact, it is the expectation of $(y - f(x_i))^2$ over $\mathcal{D}$ which we know is $\epsilon(f)$. Using this, we see:

$$= \frac{1}{N_T} \sum_{i=1}^{n} \sum_{(x_1,y_1)} P((x_1,y_1)) * ... * \epsilon(f) * ... * 1$$

$$= \frac{\epsilon(f)}{N_T} \sum_{i=1}^{n} \sum_{(x_1,y_1)} P((x_1,y_1)) \sum_{(x_2,y_2)} P((x_2,y_2)) * ... * 1 * 1 * ... * 1$$

The remaining terms now are again simply sums over a probability distribution and thus again sum to one. Finally, this yields:

$$= \frac{\epsilon(f)}{N_T} \sum_{i=1}^{n} 1$$

$$E_{train}[\hat{\epsilon}_{train}(f)] = \epsilon(f)$$

Using the exact same logic as above and assuming the data points $[(x_1, y_1), ..., (x_n, y_n)]$ with $N_{test} = n$ for our testing sample are sampled independently from $\mathcal{D}$ we can find:

$$E_{test}[\hat{\epsilon}_{test}(f)] = \epsilon(f)$$

This naturally shows that:

$$E_{test}[\hat{\epsilon}_{test}(f)] = E_{train}[\hat{\epsilon}_{train}(f)] = \epsilon(f)$$

Our last goal is to show that $E_{test}[\hat{\epsilon}_{test}(\hat{f}] = \epsilon(\hat{f})$. Since we showed above that for all fixed f it holds that $E_{test}[\hat{\epsilon}_{test}(f)] = \epsilon(f)$ all we have to do is fix $f = \hat{f}$ where $\hat{f}$ is the fit achieved from our training data. Since $\hat{f}$ is based solely on the training data, it is fixed regardless of the testing data set we use as long as the

testing data is sampled separately from the training data (as it should be). Thus:

$$E_{test}[\hat{\epsilon}_{test}(\hat{f})] = \epsilon(\hat{f})$$

Problem 3b)

No the above equation is not true for the training error. Specifically, in general $E_{train}[\hat{\epsilon}_{train}(\hat{f}_{train})] \neq E_{train}[\epsilon(\hat{f})]$. The above argument for part a) breaks down in this case as we initially assumed that our function f was fixed. Thus, when we iterated over all the training sets T in the second line (copied below for clarity/convenience) we could expect f to remain the same.

$$E_{train}[\hat{\epsilon}_{train}(f)] = \sum_{T \subset \mathcal{D}} P\left((x_1, y_1), (x_2, y_2), ..., (x_n, y_n)\right) \frac{1}{N_{train}} \sum_{(x,y) \in T} (y - f(x))^2$$

However, in the case of $\hat{f}$ we know that $\hat{f}$ depends on the training set T: it is a function chosen based on the training set to minimize the error across that training set. Thus, the function $\hat{f}$ is not the same function across all training sets and this expansion displayed above is not valid. Instead, $\hat{f}$ is a function of both the input x and the training set T. Thus, $f \rightarrow \hat{f}(x, T)$. When we make this change, the approach fails at the following line (copied below for convenience):

$$= \frac{1}{N_T} \sum_{i=1}^{n} \sum_{(x_1, y_1)} P((x_1, y_1)) * ... * \sum_{(x_i, y_i)} P((x_i, y_i))(y_i - f(x_i))^2 * ... * \sum_{(x_n, y_n)} P((x_n, y_n))$$

Since $\hat{f}$ is now a function of T, we can no longer pull f out from the inner sum and move it to the i^{th} position as we did above. This is because $\hat{f}$ depends on all datapoints (x_i, y_i) while previously f was independent of them. Thus here the proof breaks down.

Problem 3c)

Using the hint in the problem we see:

$$E_{train,test}[\hat{\epsilon}_{test}(\hat{f}_{train})] = \sum_{f \in \mathcal{F}} E_{test}[\hat{\epsilon}_{test}(f)] P_{train}(\hat{f}_{train} = f)$$

As we showed in problem 3a):

$$E_{test}[\hat{\epsilon}_{test}(f)] = E_{train}[\hat{\epsilon}_{train}(f)]$$

Combining these two findings we see that

$$\sum_{f \in \mathcal{F}} E_{test}[\hat{\epsilon}_{test}(f)] P_{train}(\hat{f}_{train} = f) = \sum_{f \in \mathcal{F}} E_{train}[\hat{\epsilon}_{train}(f)] P_{train}(\hat{f}_{train} = f)$$

We know that for any $f \in \mathcal{F}$ that $\hat{\epsilon}_{train}(\hat{f}_{train}) \leq \hat{\epsilon}_{train}(f)$. Thus it must be true that the same holds true for the expectation:

$$E_{train}[\hat{\epsilon}_{train}(\hat{f}_{train})] \leq E_{train}[\hat{\epsilon}_{train}(f)]$$

Using this we see:

$$\geq \sum_{f \in \mathcal{F}} E_{train}[\hat{\epsilon}_{train}(\hat{f}_{train})] P_{train}(\hat{f}_{train} = f)$$

Since we are summing over $f \in \mathcal{F}$ and the first term contains no f in it, we can pull it out of the sum:

$$= E[\hat{\epsilon}_{train}(\hat{f}_{train})] \sum_{f \in \mathcal{F}} P_{train}(\hat{f}_{train} = f)$$

Since in the summation we sum over the probabilities of all possible values of f, this part must sum to one.

$$= E[\hat{\epsilon}_{train}(\hat{f}_{train})]$$

Overall this yields:

$$E_{train}[\hat{\epsilon}_{train}(\hat{f}_{train})] \leq E_{train,test}[\hat{\epsilon}_{test}(\hat{f}_{train})]$$

4 Problem 4

Problem 4a)

At small values of m, the partitions are small and only include a few data points. As a result of this, there is going to be relatively high variance in the model generated as the natural variation of a single data point can have a large effect on the model generated. However, with small partitions the approximated model is going to be better at capturing local behavior in the function, causing a low bias as would be expected due to the bias-variance trade-off.

Conversely, at high values of m the partitions are very large and include many data points. As a result, the variation in a single data point has little effect on the model generated. Intuitively, we sort of expect these variations to average out over the many data points leading to low variance in the model generated (in fact the law of large numbers would suggest the mean of these random variables, the data points, approaches the expected value as the number of points increases). However, if the model is not some kind of constant function these large partitions are likely not going to capture the local dynamics of the true function very well. As a result of this, our model bias is going to be high as would be expected due to the bias-variance trade-off.

Problem 4b)

Firstly, we can note that summing over i from 1 to n is the same as summing over i from 1 to m, then over $m+1$ to $2m$, etc and adding those all together.

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

$$= \frac{1}{n}\sum_{j=1}^{n/m}\sum_{i=(j-1)m+1}^{jm}\left(E\Big[\sum_{k=1}^{n/m} c_k 1\{i \in ((k-1)m, km]\}\Big] - f(x_i)\right)^2$$

Based on our outer two summations, we can see that the indicator random variable then only takes on value 1 when $k = j$ and always takes on zero otherwise. Thus, the inner sum is redundant:

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

Next we calculate the expectation of c_j

$$E[c_j] = E\Big[\frac{1}{m}\sum_{i=(j-1)m+1}^{jm} y_i\Big]$$

$$= \frac{1}{m} \sum_{i=(j-1)m+1}^{jm} E[f(x_i) + \epsilon]$$

Since $E[\epsilon] = 0$ this leaves:

$$E[c_j] = \bar{f}^{(j)}$$

Thus overall we have:

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

Problem 4c)

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

Note that the value of $\hat{f}_m(x_i)$ is identical for all i in the same partition (i.e. $i \in \{(j-1)m+1, ..., jm\}$ for some partition j) and thus this yields:

$$= \frac{1}{n}\sum_{j=1}^{n/m} mE[(\hat{f}_m(x_i) - E[\hat{f}_m(x_i)])^2]$$

In part b) above we showed that for a fixed partition j, $\hat{f}_m(x_i) = c_j$ for all x_i in this partition. Furthermore, we showed that $E[\hat{f}_m(x_i)] = \bar{f}^{(j)}$. Thus:

$$= \frac{1}{n}\sum_{j=1}^{n/m} mE[(c_j - \bar{f}^{(j)})^2]$$

This shows the first equality. Furthermore, we can see that:

$$= \frac{1}{n}\sum_{j=1}^{n/m} mE[(\frac{1}{m^2}\sum_{i=(j-1)m+1}^{jm}(y_i - f(x_i))^2]$$

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

$$= \frac{n\sigma^2}{mn} = \frac{\sigma^2}{m}$$

Problem 4d)

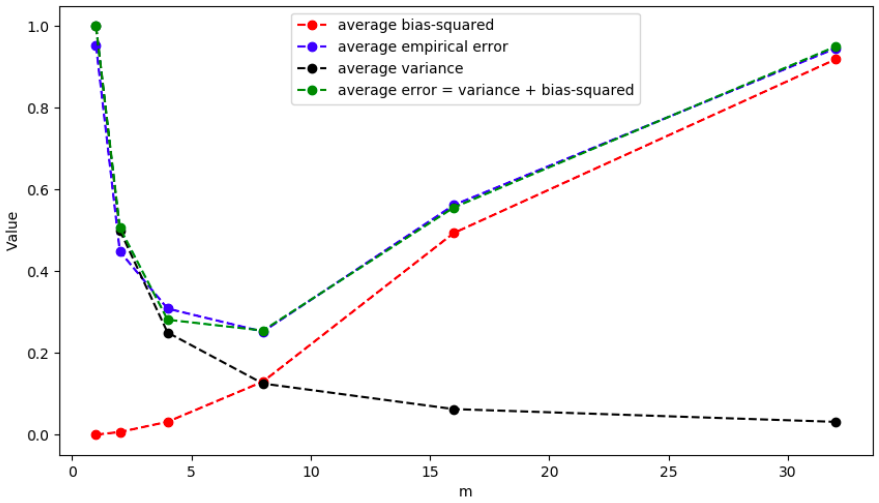

Figure 1: Average $bias^2$, variance, and empirical error versus m

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

n = 256
variance = 1
f_x = lambda x: 4*np.sin(np.pi * x)*np.cos(6*np.pi*x**2)

all_errors = []
all_biases = []
all_variances = []
all_total = []

m_vals = [1,2,4,8,16,32]

# generate the relevant random data
x_vals = [i/n for i in range(1,n+1)]
y_vals = [f_x(x) + np.random.randn() for x in x_vals] # can just
                                    use randn as variance == 1

for m in m_vals:
    empirical_error = 0
    bias_squared = 0

#    fm_vals = []

    # for each interval {1, .., m}, {m+1, ..., 2m} --> {0, 1, ...,
                                      m-1}, {m, ..., 2m-1} due to
                                      pythons
```

```
    # zero based indexing
    # indices differ by one due to python 0 based indexing

    for j in range(1, n//m +1):

        # calculate the average value of the data, y_i,  over the
                                            jth interval
        fm = 1/m * sum(y_vals[(j-1)*m : j*m]) # wont include j*m

        # calculate the true average value of the function, f(x),
                                            over the jth interval
        fj = 1/m * sum([f_x(x) for x in x_vals[(j-1)*m: j*m]])

        empirical_error += sum([(fm - f_x(x))**2 for x in x_vals[(j
                                            -1)*m : j*m]])
        bias_squared += sum([(fj - f_x(x))**2 for x in x_vals[(j-1)
                                            *m: j*m]])

#         fm_vals.extend([fm]*m)


    average_empirical_error = 1/n * empirical_error
    all_errors.append(average_empirical_error)

    average_variance = variance / m
    all_variances.append(average_variance)

    average_bias_squared = 1/n*bias_squared
    all_biases.append(average_bias_squared)

    # plot sum of the average bias and average variance
    all_total.append(average_variance + average_bias_squared)

    # TESTING CODE ========================
    '''
    plt.plot(x_vals, y_vals, 'k.')
    plt.plot(x_vals, [f_x(x) for x in x_vals], 'b-')
    plt.plot(x_vals, fm_vals, 'r-')
    plt.legend(['data', 'true f(x)', 'Estimate f_16(x)'])
    plt.show()

    time.sleep(100)
    '''
    # ====================================


plt.plot(m_vals, all_biases, 'ro--')
plt.plot(m_vals, all_errors, 'bo--')
plt.plot(m_vals, all_variances, 'ko--')
plt.plot(m_vals, all_total, 'go--')
plt.xlabel('m')
plt.ylabel('Value')
plt.legend(['average bias-squared', 'average empirical error', '
                                    average variance', 'average error
                                     = variance + bias-squared'])
plt.show()
```

4e)

By the Mean Value Theorem we know that the average $\bar{f}^{(j)}$ is between the maximum and minimum values within its partition, $f(x_i)$ for $i \in \{(j-1)m+1, ..., jm\}$

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

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

Since both $f(x_i)$ and $\bar{f}^{(j)}$ are bounded between $[f(x_\ell)_{min}, f(x_f)_{max}]$ for $\ell, f \in \{(j-1)m+1, ..., jm\}$ for all j, we know their difference is at most the difference between the max and the minimum value on the interval. Thus:

$$\leq \frac{1}{n}\sum_{j=1}^{n/m}\sum_{i=(j-1)m+1}^{jm}\left|f(x_\ell)_{min} - f(x_f)_{max}\right|^2$$

Using the fact tht f is $L - Lipschitz$ we can see that:

$$\left|f(x_\ell)_{min} - f(x_f)_{max}\right| \leq \frac{L}{n}|f-\ell| \leq \frac{L}{n}|jm - ((j-1)m+1)| = \frac{L}{n}|m-1|$$

This can be used to further simplify the above expression:

$$\leq \frac{1}{n}\sum_{j=1}^{n/m}\sum_{i=(j-1)m+1}^{jm}\left(\frac{L}{n}|m-1|\right)^2$$

$$= \frac{L^2(m-1)^2}{n^3}\sum_{j=1}^{n/m}\sum_{i=(j-1)m+1}^{jm} 1$$

$$= \frac{L^2(m-1)^2}{n^2}$$

$$\leq \frac{L^2m^2}{n^2}$$

Thus the average bias-squared is $\leq \frac{L^2m^2}{n^2} \in O\left(\frac{L^2}{m^2}n^2\right)$. Thus the bias-squared is $O\left(\frac{L^2m^2}{n^2}\right)$

Now consider the fact that the total error behaves $O\left(\frac{L^2m^2}{n^2} + \frac{1}{m}\right)$. We can minimize this with respect to m:

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

Thus, at this optmized value of m we can see the total error behaves as:

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

5 Problem 5

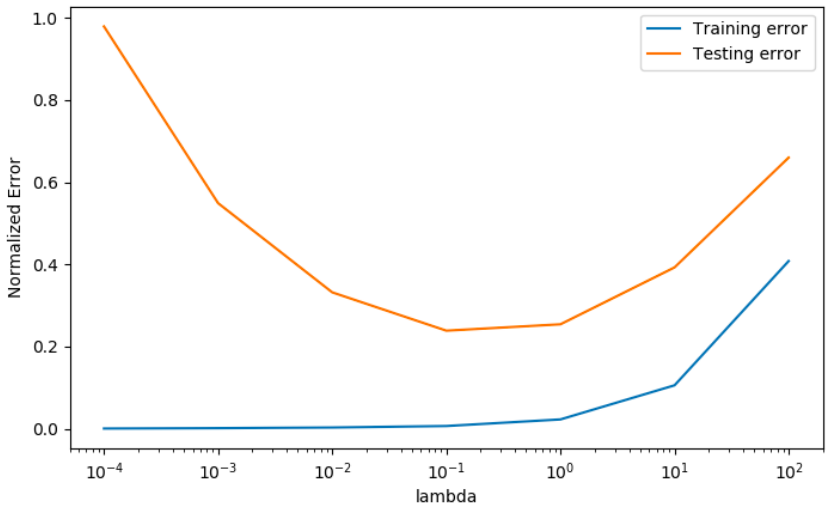

Figure 2: Training and Test error as a function of the L_2 regularization parameter λ in Ridge Regression

As λ grows, the $\lambda||w||^2$ term has a larger and larger influence on the solution of the least squares system. Initially as λ is low the solution is roughly similar to the Least Squares. As a result, it is prone to overfitting the training data. This is why the training error is so low when λ is small. Similarly, due to the overfitting the solution $\hat{w}$ poorly generalizes to data outside the training set. This is why the test error is so high with small λ. However, as λ grows it begins pushing $\hat{w}$ closer and closer to zero. This helps discourage overfitting by filtering out solutions with large coefficients which result in large L_2 norms. These large coefficients generate a lot of local variation in the solution which often results in overfitting, so filtering against them helps generate more generalizable solutions. As long as λ is not too large, the optimal Ridge Regression solution still requires a low Least Squared Error, resulting in a low training error, and the increased generalizability leads to a low testing error. This would be where we see the bottom of the bowl in the testing error. Lastly, if we increase λ too far the $\lambda||w||^2$ term has too large of an effect on the solution. As a result, $\hat{w}$ is pushed towards zero regardless of how well it performs in the least squares sense. This leads to a dramatic increase in the training error as $\hat{w}$ cannot explain the dynamics of the system. Correspondingly, $\hat{w}$ is not generalizable and leads to a high testing error. This is why both the training and testing error increase when λ gets eventually too large.

```
# Problem 5

import numpy as np
import matplotlib.pyplot as plt

num_trials = 30

lam_vals = [0.0001, 0.001, 0.01, 0.1, 1, 10, 100] # lambda values

# These will be 2D arrays with each row representing a given lambda
                                    value and
# each column a trial at that lambda value
all_train_errors = np.zeros((len(lam_vals), num_trials))
all_test_errors = np.zeros((len(lam_vals), num_trials))

for j in range(num_trials): # run 30 trials
    train_n = 100
    test_n = 1000
    d = 100
    X_train = np.random.normal(0,1, size=(train_n,d))
    a_true = np.random.normal(0,1, size=(d,1))
    y_train = X_train.dot(a_true) + np.random.normal(0,0.5,size=(
                                        train_n,1))
    X_test = np.random.normal(0,1, size=(test_n,d))
    y_test = X_test.dot(a_true) + np.random.normal(0,0.5,size=(
                                        test_n,1))

    # lam = lambda values
    for i, lam in enumerate(lam_vals):
        Xt_X_plus_lambda = np.dot(X_train.T, X_train) + lam * np.
                                            eye(d,d)
        Xt_y = np.dot(X_train.T, y_train)
        w_hat = np.linalg.solve(Xt_X_plus_lambda, Xt_y)

        test_error = np.linalg.norm(np.dot(X_test, w_hat) - y_test)
                                            / np.linalg.norm(y_test)
        train_error = np.linalg.norm(np.dot(X_train, w_hat) -
                                            y_train) / np.linalg.norm
                                            (y_train)

        all_test_errors[i, j] = test_error
        all_train_errors[i,j] = train_error

# calculate the average along each row (each lambda value)
ave_train_errors = np.mean(all_train_errors, axis=1)
ave_test_errors = np.mean(all_test_errors, axis=1)

plt.semilogx(lam_vals, ave_train_errors)
plt.semilogx(lam_vals, ave_test_errors)
plt.legend(['Training error', 'Testing error'])
plt.ylabel('Normalized Error')
plt.xlabel('lambda')
plt.show()
```

6 Problem 6

6a)

$$\sum_{j=0}^{k} \Big[||Xw_j - Ye_j||^2 + \lambda ||w_j||^2 \Big]$$

$$\sum_{j=0}^{k} \Big[(Xw_j - Ye_j)^T (Xw_j - Ye_j) + \lambda (w_j)^T (w_j) \Big]$$

$$\frac{\delta C}{\delta w_j} = 0 = 2X^T(Xw_j - Ye_j) + \lambda 2 w_j$$

$$0 = 2X^T X w_j - 2X^T Y e_j + 2\lambda I w_j$$

$$0 = (X^T X + \lambda I) w_j - X^T Y e_j$$

$$(X^T X + \lambda I) w_j = X^T Y e_j$$

Note that $X^T X$ is positive semi-definite for any matrix X. We can see this by:

$$a^T X^T X a = (Xa)^T (Xa) = ||Xa||^2 \geq 0$$

Furthermore, λI is clearly positive definite for $\lambda > 0$:

$$a^T \lambda I a = \lambda a^T a = \lambda ||a||^2 > 0 \qquad \forall a \neq 0, \lambda > 0$$

Thus the sum $X^T X + \lambda I$ is positive definite and is thus invertible as its eigenvalues are all positive.

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

From here, we can construct the full $\hat{W}$ matrix as we have an equation for each column:

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

6b)

Training Error = 0.14805
Testing Error = 0.1466

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


# Problem 6b)

def train(X, Y, lam):
    '''
    Given training data X and its associated labels Y, performs
                                    ridge regression
    using the given regularization parameter lambda = lam. Returns
                                    the weights matrix
    W_hat generated by this procedure.
    '''
    d = X.shape[1]

    Xt_X_plus_lambda = np.dot(X.T, X) + lam * np.eye(d,d)
    Xt_Y = np.dot(X.T, Y)

    return np.linalg.solve(Xt_X_plus_lambda, Xt_Y)

def predict(W, X_prime):
    '''
    Given a trained mapping W from pixelspace to digitspace and new
                                     observations X, makes
                                    predictions
    about the digit present in each image in X. Returns these
                                    labels (as digits)
    '''
    return np.argmax(np.dot(X_prime, W), axis = 1)

def check_error(predictions, actual):
   '''
   Given two label vectors (as digits not one-hot), true labels and
                                    predicted labels, calculates
                                   the error rate.
   '''
   if len(predictions) != len(actual):
       raise ValueError("Two label vectors must be the same length"
                                        )

   return sum([1 for i in range(len(predictions)) if predictions[i]
                                    != actual[i] ]) / len(
                                   predictions)


# ======= LOAD MNIST DATA ==================================
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

# Convert training labels to one hot
# i.e. encode an i as [0, 0, ..., 0, 1, 0, ... 0] where only the
                                    ith entry is nonzero
Y_train = np.zeros((X_train.shape[0], 10))
for i,digit in enumerate(labels_train):
    Y_train[i, digit] = 1


num_training_images = X_train.shape[0]
num_test_images = X_test.shape[0]
d = X_train.shape[1]
# ============================================================

W = train(X_train, Y_train, 10**(-4))
train_predictions = predict(W, X_train)
test_predictions = predict(W, X_test)

test_error = check_error(test_predictions, labels_test)
train_error = check_error(train_predictions, labels_train)

print("Training error: ", train_error, "\nTest error: ", test_error
                                    )
```

6c)

The code below relies on the train, predict, and check_error used in 6b) above as well as the code that loads the MNIST data and converts to one-hot encoding. I tested values up to of the hyperparameter up to $p = 10000$. After this point, my laptop was having trouble running the computations. However, if we were to continue these computations we would eventually expect the testing error to begin increasing again.

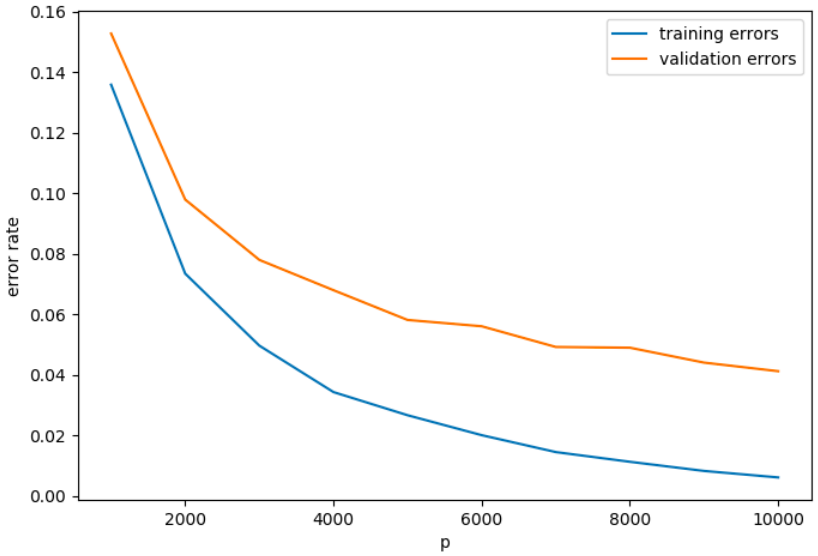

Figure 3: Training/Validation Error versus the hyperparameter p

```
# Problem 6c)
variance = 0.1
lam = 10**(-4) # lambda for ridge regression
fraction_train = 0.8

#p_vals = [500, 1000, 2000, 3000, 4000, 5000, 6000, 8000, 10000,
                                    20000]
p_vals = [2000*i for i in range(1, 8)]
all_train_errors = []
all_validation_errors = []
#all_test_errors = []

# choose which images/indices will be train/validation data
shuffled_indices = np.arange(num_training_images)
np.random.shuffle(shuffled_indices)
train_indices = shuffled_indices[0:int(fraction_train *
```

```
                                    num_training_images)]
validation_indices = shuffled_indices[int(fraction_train *
                                    num_training_images) : ]

# generate the corresponding labels based on the shuffled indices
Yp_train = Y_train[train_indices, :]
labels_train_p = labels_train[train_indices]
labels_validate_p = labels_train[validation_indices]

for p in p_vals:

    print("Running using p = ", p)
    # calculate G and b
    G = np.random.normal(0, np.sqrt(variance), size = (p,d))
    b = np.random.uniform(low=0, high=2*np.pi, size=(p,1))

    # transform each row x_i of X by cos(Gx_i + b) --> new_x_i =
                                        cos(x_i^TG^T + b^T) --> X_new
                                         = cos(XG^T + b^T)
    X_train_transformed = np.cos(np.dot(X_train, G.T) + b.T)


    # selected out the appropriate images for training/validation
                                        from pre-selected
    # indices
    Xp_train = X_train_transformed[train_indices, :]
    Xp_validate = X_train_transformed[validation_indices, :]

    Wp = train(Xp_train, Yp_train, lam)
    train_predictions = predict(Wp, Xp_train)
    validation_predictions = predict(Wp, Xp_validate)

    train_error = check_error(train_predictions, labels_train_p)
    all_train_errors.append(train_error)
    validation_error = check_error(validation_predictions,
                                        labels_validate_p)
    all_validation_errors.append(validation_error)

    print("training error: ", train_error)
    print("validation error: ", validation_error)


# Plot the results!
plt.plot(p_vals, all_train_errors)
plt.plot(p_vals, all_validation_errors)
plt.legend(["training errors", "validation errors"])
plt.xlabel('p')
plt.ylabel('error rate')
plt.show()
```

6d)

For each (x_i, y_i) in the training dataset, define an indicator random variable that represents whether the i^{th} data point (image) is correctly classified

$$1_{test}\{\hat{f}(x_i) == y_i\}$$

Using this definition, it is clear that the testing error can be defined as follows:

$$\hat{\epsilon}_{test}(\hat{f}) = \frac{1}{N_{test}} \sum_{i=1}^{N_{test}} 1_{test}\{\hat{f}(x_i) == y_i\}$$

Here we can see:

$$1_{test}\{\hat{f}(x_i) == y_i\} \in \{0, 1\} \subset [0, 1]$$
$$E[1_{test}\{\hat{f}(x_i) == y_i\}] = E[\hat{\epsilon}_{test}(\hat{f})] = \epsilon(\hat{f})$$

Where the final equality we showed in problem 3a) as $\hat{f}$ is fixed over the test set. Lastly, since our datapoints are assumed to be sampled randomly from the underlying distribution, these variables are iid. Thus our indicator random variable satisfies all constraints of Hoeffding's Inequality.

$$P\left(\left|\left(\frac{1}{m}\sum_{i=1}^{m} X_i\right) - \mu\right| \leq \sqrt{\frac{(b-a)^2 ln(2/\delta)}{2m}}\right) \leq \delta$$

If we want a 95% confidence interval simply choose $\delta = 0.05$. Set $m = N_{test}, a = 0, b = 1, X_i = 1_{test}\{\hat{f}(x_i) == y_i\}$ and we can see:

$$P\left(\left|\left(\frac{1}{N_{test}}\sum_{i=1}^{N_{test}} 1_{test}\{\hat{f}(x_i) == y_i\}\right) - E[1_{test}\{\hat{f}(x_i) == y_i\}]\right| \leq \sqrt{\frac{ln(40)}{2N_{test}}}\right) \leq 0.05$$

$$P\left(\left|\hat{\epsilon}_{test}(\hat{f}) - \epsilon(\hat{f})\right| \leq \sqrt{\frac{ln(40)}{2N_{test}}}\right) \leq 0.05$$

Thus, with 95% confidence we can see that:

$$\hat{\epsilon}_{test}(\hat{f}) - \sqrt{\frac{ln(40)}{2N_{test}}} \leq \epsilon(\hat{f}) \leq \hat{\epsilon}_{test}(\hat{f}) + \sqrt{\frac{ln(40)}{2N_{test}}}$$

Using the optimal value of $\hat{p} = 10,000$ we determined from part 6c) we can build a confidence interval for the true classification error $\epsilon(\hat{f})$. From the MNIST database, we see $N_{test} = 10000$ and the code below generates a test error value of $\hat{\epsilon}_{test}(\hat{f}) = 0.0501$ giving the 95% confidence interval for $\epsilon(\hat{f})$

$$(0.036518984842593805, 0.06368101515740619)$$

The code below relies on the predict, train, check_errors, and load MNIST functions shown above in 6*b*)

```
# Problem 6d)

# \hat{epsilon}(\hat{f})

X_test_transformed = np.cos(np.dot(X_test, G.T) + b.T)
test_error = check_error(predict(Wp_best,X_test_transformed),
                                    labels_test)
N_test = num_test_images
square_root_part = np.sqrt(np.log(40) /  (2*N_test))

print("N_test: ", N_test)
print("best_p: ", best_p)
print("test error: ", test_error)
print("Square root part: ", square_root_part)


print("True classification error 95% confidence interval: (", \
        test_error - square_root_part, ", ", test_error +
                                          square_root_part, ")")
```