Homework #1

Spring 2020, CSE 546: Machine Learning

Roman Levin
1721898

Collaborators: compared answers with Tyler Chen, Diya Sashidhar, Katherine Owens

Problems A

Conceptual Questions

A.0 **Solution:**

- **Bias** – how far our expected model is from the best possible estimator. Or, in expectation (over all possible data), how well our model fits the best possible estimator.

  Variance – how much on average our model changes (how much the predictions vary) around our expected model over iid resampling of the data from the underlying generating distribution. (That is, every time we draw a new dataset from the underlying distribution and refit the model.)

  Bias-Variance Tradeoff – The true total expected error of a model decomposes into irreducible error and learning error. The learning error decomposes into the bias squared and variance squared. So, we could reduce the learning error by reducing the bias and/or the variance of our model. However, usually models with lower bias have higher variance and vice versa, so there is a tradeoff between bias and variance. We would like to find the sweet spot, that is, we want to balance bias and variance to get smaller learning error.

- As the model **complexity increases, bias decreases and variance increases**. As the model **complexity decreases, bias increases and variance decreases**.
- **False**
- **True** (Or True-ish. Usually, using more data has a regularizing effect, so more data reduces variance, especially if the model is expressive enough to capture the "truth". However, we could also think of artificial counter-examples like a model which has no trainable parameters and just outputs 42 for any input. For this model, the variance will be always zero, but we would probably prefer models with trainable parameters.)
- **False** (e.g. using zero features does not result in better models)
- **Train set** (we should NEVER use the test set for tuning the model)
- **False**

MLE

A.1 **Solution:**

a. Let's find the ML estimator for an arbitrary n: $\lambda_{MLE} = \arg\min_\lambda L(\lambda)$.

- $L(\lambda) = -\log \prod_{i=1}^n \text{Poi}(x_i|\lambda) = -\sum_{i=1}^n (-\lambda + x_i \log\lambda - \log x_i!)$
- $L'(\lambda) = n - \sum_i x_i/\lambda = 0 \Leftrightarrow \boxed{\lambda_{MLE} = \frac{\sum_{i=1}^n x_i}{n}}$
- (Check $L''(\lambda) = \sum_i x_i/\lambda^2 \geq 0$ since $x_i \geq 0$. But if at least one of $x_i > 0$, then $L''(\lambda) > 0$, so we indeed found the minimum. If $\forall x_i = 0$, then $\lambda_{MLE} = 0$ which is also consistent with the formula above. $\square$
  The answer for part a. is then:

$$\boxed{\lambda_{MLE} = \frac{\sum_{i=1}^5 x_i}{5} = 6/5}$$

b.

$$\boxed{\lambda_{MLE} = \frac{\sum_{i=1}^6 x_i}{6} = 10/6 = 5/3}$$

c.

$$\boxed{\lambda_{MLE,5} = 6/5, \;\; \lambda_{MLE,6} = 5/3}$$

A.2 **Solution:**
Notation: $\mathbb{1}\{\cdot\}$ stands for an indicator function.

- The likelihood: $L(\theta) = \prod_{i=1}^n \frac{1}{\theta}\mathbb{1}\{x_i \in [0,\theta]\}$. We want to find θ_{MLE} which maximizes $L(\theta)$.
- Note that $\exists x_i \notin [0,\theta] \Rightarrow L(\theta)$, otherwise $L(\theta) = \frac{1}{\theta^n} > 0$ since $\theta \geq 0$. Note also that decreasing θ increases $\frac{1}{\theta^n}$. So we need to choose the smallest possible θ, s.t. $\forall i : x_i \in [0,\theta]$.
- The smallest possible θ, s.t. $\forall i : x_i \in [0,\theta]$ is

$$\boxed{\theta_{MLE} = \max_i x_i}$$

Overfitting

A.3 **Solution:**

a.

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

Where the third equality follows from the fact that the points in S_{train} are iid, and that sampling the entire S_{train} iid and picking the points from it one by one is equivalent to just sampling one point from the

generating distribution over and over again N_{train} times (provided that f is independent of S_{train}, because otherwise we cannot assume that $(f(x)-y)^2$ are iid in $\sum_{(x,y)\in S_{\text{train}}}(f(x)-y)^2$). f is indeed independent of the training set by the problem statement).

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

Where the third equality follows from the fact that the points in S_{test} are iid, and that sampling the entire S_{test} iid and picking the points from it one by one is equivalent to just sampling one point from the generating distribution over and over again N_{test} times (provided that f is independent of S_{test}, because otherwise we cannot assume that $(f(x)-y)^2$ are iid in $\sum_{(x,y)\in S_{\text{test}}}(f(x)-y)^2$). f is indeed independent of the test set by the problem statement).

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

Where the third equality follows from the fact that the points in S_{test} are iid, and that sampling the entire S_{test} iid and picking the points from it one by one is equivalent to just sampling one point from the generating distribution over and over again N_{test} times (provided that $\widehat{f}$ is independent of the S_{test}, because otherwise we cannot assume that $(\widehat{f}(x)-y)^2$ are iid in $\sum_{(x,y)\in S_{\text{test}}}(\widehat{f}(x)-y)^2$). Since $\widehat{f}$ is found using the training set only, it is indeed independent of the test set.).

b. **No**, in general it is not true. The third equality in the above arguments would break down because $\widehat{f}$ is not independent of S_{train} and thus we cannot assume that $(\widehat{f}(x)-y)^2$ are iid in $\sum_{(x,y)\in S_{\text{train}}}(\widehat{f}(x)-y)^2$.

c. First, note that:

$$\widehat{\epsilon}_{\text{train}}(\widehat{f}_{\text{train}}) \leq \widehat{\epsilon}_{\text{train}}(f)\forall f\in\mathcal{F} \Rightarrow \mathbb{E}_{\text{train}}[\widehat{\epsilon}_{\text{train}}(\widehat{f}_{\text{train}})] \leq \mathbb{E}_{\text{train}}[\widehat{\epsilon}_{\text{train}}(f)].$$

Now, using the hint, part a., the fact that $\widehat{f}_{\text{train}}$ minimizes the training error, and that pmf sums to one over all possible outcomes,

$$\mathbb{E}_{\text{train,train}}[\widehat{\epsilon}_{\text{test}}(\widehat{f}_{\text{train}})] = \sum_{f\in\mathcal{F}}\mathbb{E}_{\text{train,test}}[\widehat{\epsilon}_{\text{test}}(f)\mathbb{1}\{\widehat{f}_{\text{train}}=f\}] = \sum_{f\in\mathcal{F}}\mathbb{E}_{\text{test}}[\widehat{\epsilon}_{\text{test}}(f)]\mathbb{E}_{\text{train}}[\mathbb{1}\{\widehat{f}_{\text{train}}=f\}]$$
$$= \sum_{f\in\mathcal{F}}\mathbb{E}_{\text{test}}[\widehat{\epsilon}_{\text{test}}(f)]\mathbb{P}_{\text{train}}[\widehat{f}_{\text{train}}=f] = \sum_{f\in\mathcal{F}}\mathbb{E}_{\text{train}}[\widehat{\epsilon}_{\text{train}}(f)]\mathbb{P}_{\text{train}}[\widehat{f}_{\text{train}}=f] \geq$$
$$\geq \sum_{f\in\mathcal{F}}\mathbb{E}_{\text{train}}[\widehat{\epsilon}_{\text{train}}(\widehat{f}_{\text{train}})]\mathbb{P}_{\text{train}}[\widehat{f}_{\text{train}}=f] = \mathbb{E}_{\text{train}}[\widehat{\epsilon}_{\text{train}}(\widehat{f}_{\text{train}})]\sum_{f\in\mathcal{F}}\mathbb{P}_{\text{train}}[\widehat{f}_{\text{train}}=f] = \mathbb{E}_{\text{train}}[\widehat{\epsilon}_{\text{train}}(\widehat{f}_{\text{train}})]$$

$\square$

Bias-Variance Tradeoff

B.1 **Solution:**

a. We could view $1/m$ as a proxy for the complexity of the model. That is, **for large** m (i.e. $m=n$, simple model), the estimator would just predict the average of all the observations, so **I would expect the model to have high bias but low variance**. On the other hand, **for small** m (i.e. $m=1$, high complexity, the model would have only one observation per interval and would predict its value for any input within that interval), so **I would expect the model to have low bias but high variance**.

b. First, observe that for x_i within j-th interval, all the indicators in the definition of $\widehat{f}_m$ are zero except for the one corresponding to the j-th interval, so we have:

$$\mathbb{E}[\widehat{f}_m(x_i)] = \mathbb{E}[c_j] = \mathbb{E}[\frac{1}{m}\sum_{k=(j-1)m+1}^{jm} y_k] = \frac{1}{m}\sum_{k=(j-1)m+1}^{jm} \mathbb{E}[y_k] = \frac{1}{m}\sum_{k=(j-1)m+1}^{jm} f(x_k) = \bar{f}^{(j)}.$$

Now, using this and splitting the sum over all the points into double sum where the outer summation is over the intervals and the inner summation is over the points in a given interval, we obtain:

$$\frac{1}{n}\sum_{i=1}^{n}(\mathbb{E}[\widehat{f}_m(x_i)] - f(x_i))^2 = \frac{1}{n}\sum_{j=1}^{n/m}\sum_{i=(j-1)m+1}^{jm}(\mathbb{E}[\widehat{f}_m(x_i)] - f(x_i))^2 = \frac{1}{n}\sum_{j=1}^{n/m}\sum_{i=(j-1)m+1}^{jm}(\bar{f}^{(j)} - f(x_i))^2. \qquad \square$$

c. First, since y_i are independent, observe that

$$\mathbb{V}[c_j] = \mathbb{V}[\frac{1}{m}\sum_{k=(j-1)m+1}^{jm} y_k] = \frac{1}{m^2}\sum_{k=(j-1)m+1}^{jm} \mathbb{V}[y_k] = \frac{\sigma^2}{m}.$$

Also observe that for x_i within j-th interval, all the indicators in the definition of $\widehat{f}_m$ are zero except for the one corresponding to the j-th interval, so we have: $\widehat{f}_m(x_i) = c_j$. Now, similarly with the previous part, splitting the sum over all the points into double sum where the outer summation is over the intervals and the inner summation is over the points in a given interval, we obtain:

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

$$= \frac{1}{n}\sum_{j=1}^{n/m}\sum_{i=(j-1)m+1}^{jm}\mathbb{E}[(c_j - \mathbb{E}[c_j])^2] = \boxed{\frac{1}{n}\sum_{j=1}^{n/m} m\mathbb{E}[(c_j - \bar{f}^{(j)})^2]} = \frac{1}{n}\frac{n}{m}m\mathbb{V}[c_j] = \boxed{\frac{\sigma^2}{m}}. \qquad \square$$

d. See Figure 1. The code is below, after the figure.

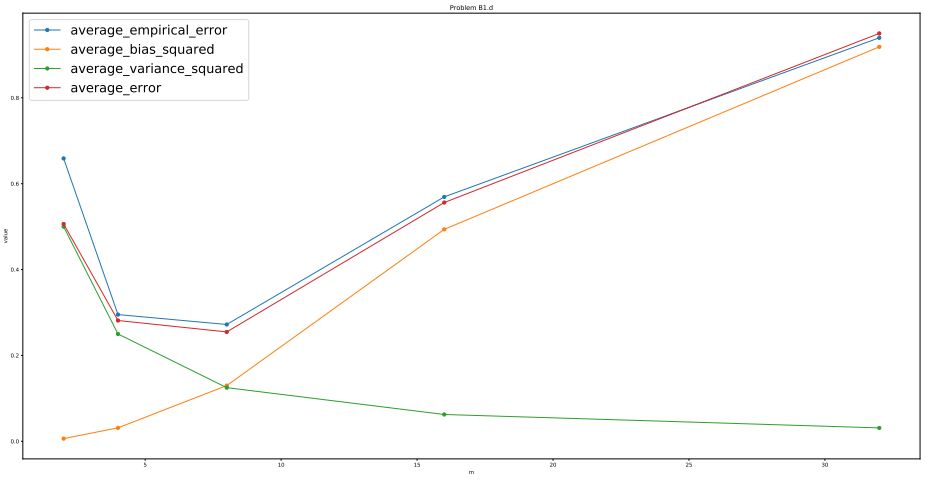

Figure 1: Problem B1.d

```
######################
#Problem B1.d, HW1
######################
import numpy as np
import matplotlib.pyplot as plt


# def indicator(x, left, right):
#         return (x > left) and (x<right)

def f_hat(x, Y, m):
    n = Y.shape[0]
    c = np.array([np.mean(Y[j*m:(j+1)*m]) for j in range(n//m)])
    x_interval_idx = x/(m/n) - 0.001
    x_interval_idx = x_interval_idx.astype('int') #Compute the interval idx
    return c[x_interval_idx]

def true_f(x):
    return 4*np.sin(np.pi*x)*np.cos(6*np.pi*x**2)

def mse(x, y):
        return np.mean((x-y)**2)

def bias_squared(X, f, m):
        n = X.shape[0]
        f_arr = f(X)
        fbar = np.array([np.mean(f_arr[j*m:(j+1)*m]) for j in range(n//m)])
        bias_squared = []
        for j in range(n//m):
                for i in range(j*m, (j+1)*m):
                        bias_squared.append((fbar[j] - f_arr[i])**2)
        return np.mean(bias_squared)

def variance_squared(sigma, m):
        return sigma**2/m


if __name__ == "__main__":
        n = 256
        sigma = 1
        eps = np.random.normal(0, sigma, n)
        X = np.arange(1,n+1)/n
        Y = true_f(X) + eps
        m = 2**np.arange(1,5+1)
        empirical_mse_arr = np.array([mse(f_hat(X, Y, mi),true_f(X)) for mi in m])
        bias_squared_arr = np.array([bias_squared(X, true_f, mi) for mi in m])
        variance_squared_arr = np.array([variance_squared(sigma, mi) for mi in m])
        avg_error_arr = bias_squared_arr + variance_squared_arr


        plt.figure(figsize = (30, 15))
        plt.plot(m, empirical_mse_arr, '-o', label = 'average_empirical_error')
        plt.plot(m, bias_squared_arr, '-o', label = 'average_bias_squared')
        plt.plot(m, variance_squared_arr, '-o', label = 'average_variance_squared')
        plt.plot(m, avg_error_arr, '-o', label = 'average_error')
```

```
plt.title('Problem B1.d')
plt.xlabel('m')
plt.ylabel('value')
plt.legend(prop={'size': 24})
plt.savefig('b1_d.pdf')
plt.show()
```

Code for B1.d

e. From b., we know that the average bias-squared is

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

To obtain an upper bound on it, first note that because of the mean-value theorem and depending on x_i, one of the two cases is correct for every term in the sum corresponding to a given interval $\sum_{i=(j-1)m+1}^{jm}(\bar{f}^{(j)}-f(x_i))^2$:

- $(\bar{f}^{(j)}-f(x_i))^2 \leq (\min_{i=(j-1)m+1,\dots,jm} f(x_i)-f(x_i))^2$
- $(\bar{f}^{(j)}-f(x_i))^2 \leq (\max_{i=(j-1)m+1,\dots,jm} f(x_i)-f(x_i))^2$

Now, we can have the same bound for both cases:

- $(\bar{f}^{(j)}-f(x_i))^2 \leq (\min_{i=(j-1)m+1,\dots,jm} f(x_i)-f(x_i))^2 \leq \frac{L^2}{n^2}(\arg\min_{i=(j-1)m+1,\dots,jm} f(x_i)-i)^2 \leq \frac{L^2}{n^2}(m-1)^2$
- $(\bar{f}^{(j)}-f(x_i))^2 \leq (\max_{i=(j-1)m+1,\dots,jm} f(x_i)-f(x_i))^2 \leq \frac{L^2}{n^2}(\arg\max_{i=(j-1)m+1,\dots,jm} f(x_i)-i)^2 \leq \frac{L^2}{n^2}(m-1)^2$

So $\sum_{i=(j-1)m+1}^{jm}(\bar{f}^{(j)}-f(x_i))^2 \leq m\frac{L^2}{n^2}(m-1)^2 \sim O(\frac{L^2m^3}{n^2})$. Thus

$$\boxed{\frac{1}{n}\sum_{j=1}^{n/m}\sum_{i=(j-1)m+1}^{jm}(\bar{f}^{(j)}-f(x_i))^2 \sim O(\frac{1}{n}\frac{n}{m}\frac{L^2m^3}{n^2}) \sim O(\frac{L^2m^2}{n^2})} \quad \square$$

So, using the expression for average variance above, we see that the total error behaves like $O(\frac{L^2m^2}{n^2}+\frac{\sigma^2}{m})$. Let's minimize this expression with respect to m:

$$\frac{d}{dm}(\frac{L^2m^2}{n^2}+\frac{\sigma^2}{m}) = 2\frac{L^2m}{n^2}-\frac{\sigma^2}{m^2} = 0 \Leftrightarrow \boxed{m=\sqrt[3]{\frac{\sigma^2n^2}{2L^2}}}$$

Plugging this value back to the error gives that the total error $\sim O\left(\frac{\sigma^{\frac{4}{3}}L^{\frac{2}{3}}}{n^{\frac{2}{3}}}\right)$. Both behaviours are intuitive:

- Increasing n, the amount of data, decreases the error and increases m so we can use simpler models
- Increasing σ, the amount of noise, increases the error and increases m and it makes sense to use simpler models in the presence of high noise to avoid fitting the noise.
- Decreasing L, making the underlying "truth" more smooth, decreases the error and increases m so we can use simpler models (e.g. the smoothest possible function is a constant and it can be learned with very simple models).

Polynomial Regression

A.4 **Solution:**
Bigger λ results in more regularization, as we can see from the plots in Figure 2. See the code for both A.4 in A.5 after Figure 3

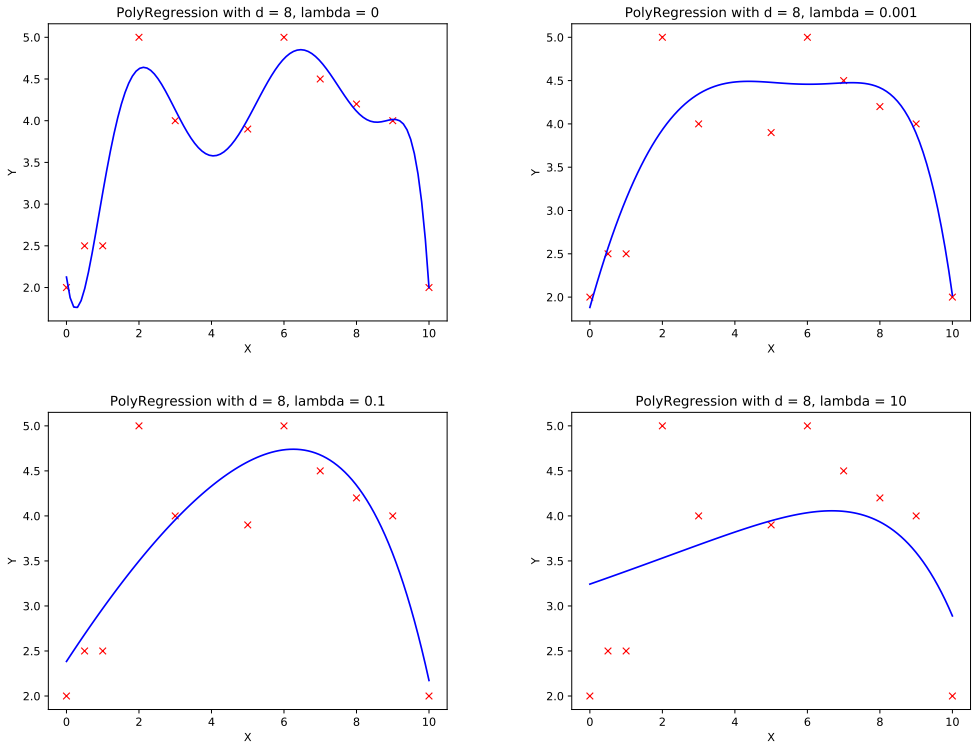

Figure 2: Problem A4

A.5 **Solution:**
See the function learningCurve in the following script:

```
##########################
#Problems A4 and A5, HW1
##########################

'''
    Using template for polynomial regression
    by Eric Eaton, Xiaoxiang Hu
'''

import numpy as np


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

class PolynomialRegression:
```

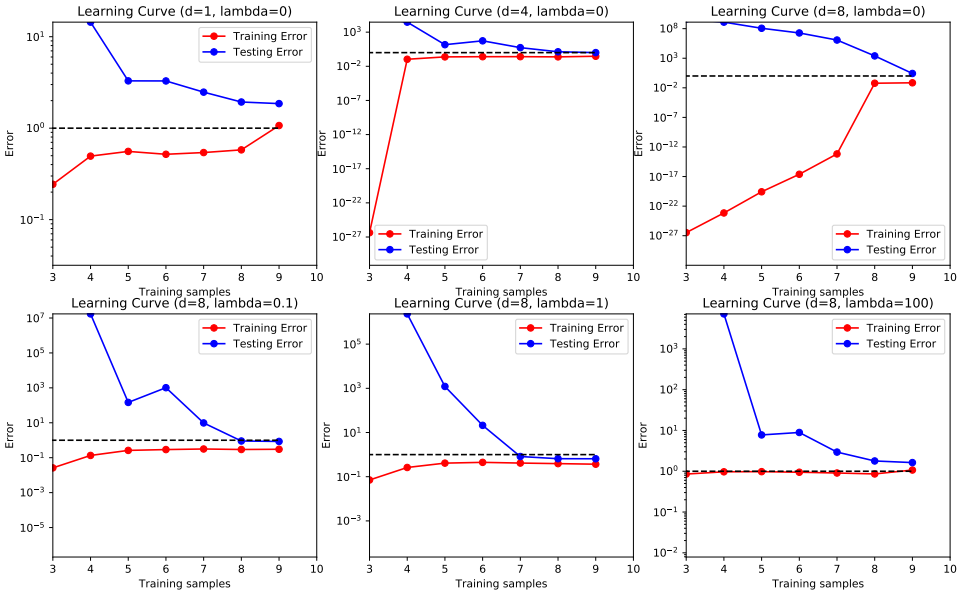

Figure 3: Problem A5

```
def __init__(self, degree=1, reg_lambda=1E-8):
    """
    Constructor
    """
    self.regLambda = reg_lambda
    self.degree = degree

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 d in range(1, self.degree):
        X_ = np.c_[X_, X**d]

    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 = X.shape[0]
        # polynomial expansion
        X_ = self.polyfeatures(X, self.degree)
        # normalization (no normalization if n = 1)
        if n!=1:
            self.mean = X_.mean(axis = 0)
            self.std = X_.std(axis = 0)
        else:
            self.mean = 0
            self.std = 1

        X_ = (X_ - self.mean)/self.std
        # adding ones
        X_ = np.c_[np.ones([n,1]), X_]

        # closed form of regression
        _, d = X_.shape
        # construct reg matrix
        reg_matrix = self.regLambda * np.eye(d)
        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 = X.shape[0]
        X_ = self.polyfeatures(X, self.degree)
        # normalize
        X_ = (X_ - self.mean)/self.std
        # adding ones
        X_ = np.c_[np.ones([n,1]), X_]

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

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



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

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

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

    n = len(Xtrain)

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

    # Here we start from two observations since we do not care about
    # the performance of the model trained with only 1 data point.

    for i in range(1,n):
        Xtrain_i = Xtrain[0:i+1]
        Ytrain_i = Ytrain[0:i+1]

        # training the model
        model = PolynomialRegression(degree=degree, reg_lambda=reg_lambda)
        model.fit(Xtrain_i, Ytrain_i)

        # make predictions
        predictions_train_i = model.predict(Xtrain_i)
        predictions_test_i = model.predict(Xtest)

        # compute errors
        error_train_i = np.mean((predictions_train_i -Ytrain_i)**2)
        error_test_i = np.mean((predictions_test_i -Ytest)**2)
        errorTrain[i] = error_train_i
        errorTest[i] = error_test_i

    return errorTrain, errorTest
```

Ridge Regression on MNIST

A.6 **Solution:**

a. As per the hint:

$$\begin{aligned}\sum_{i=1}^{n}\|W^Tx_i-y_i\|_2^2+\lambda\|W\|_F^2 &= \sum_{j=0}^{k}\left[\sum_{i=1}^{n}(e_j^TW^Tx_i-e_j^Ty_i)^2+\lambda\|We_j\|_F^2\right]\\ &= \sum_{j=0}^{k}\left[\sum_{i=1}^{n}(w_j^Tx_i-e_j^Ty_i)^2+\lambda\|w_j\|^2\right]\\ &= \sum_{j=0}^{k}\left[\|Xw_j-Ye_j\|^2+\lambda\|w_j\|^2\right]\end{aligned}$$

To minimize this quantity w.r.t. W we take the derivative and set it to zero:

$$\frac{\partial}{\partial w_j}\sum_{j=0}^{k}\left[\|Xw_j-Ye_j\|^2+\lambda\|w_j\|^2\right]=\frac{\partial}{\partial w_j}\left[\|Xw_j-Y_{:,j}\|^2+\lambda\|w_j\|^2\right]=$$
$$=X^T(Xw_j-Y_{:,j})+\lambda w_j=0\Leftrightarrow\widehat{w}_j=(X^TX+\lambda I)^{-1}Y_{:,j}$$

where $Y_{:,j}$ stands for j-th column of Y (in python-like notation since y_i are reserved for the ows of Y). That is,

$$\boxed{\widehat{W}=(X^TX+\lambda I)^{-1}X^TY}$$

The ridge regression problem is evidently convex in $\widehat{W}$ (since the Hessian is $X^TX+\lambda I\succ 0$ for $\lambda>0$) and thus we indeed have the global minimum. □

b. **Train error: 0.14805000000000001, Test error: 0.14659999999999995**. See the code below.

```
######################
#Problem A6.b, HW1
######################
import numpy as np
from mnist import MNIST

def load_dataset():
    mndata = 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

def one_hot_encoder(labels_train):
    """
    Performs one-hot encoding of the labels

    Parameters
    ----------
```

```
    labels_train : np.array of shape (n,)
        Labels

    Returns
    -------
    np.array of shape (n,k)
        One-hot-encoded labels
    """
    return np.eye(len(set(labels_train)))[labels_train]

def accuracy_error(y_true, y_pred):
    """
    Trains ridge regression using closed-form solution

    Parameters
    ----------
    y_true : np.array of shape (m,)
        Vector of true labels
    y_pred : np.array of shape (m,)
        Vector of predicted labels

    Returns
    -------
    float
        error: 1-accuracy
    """
    return 1-np.mean(y_true == y_pred)

def train(X, Y, reg_lambda):
    """
    Trains ridge regression using closed-form solution

    Parameters
    ----------
    X : np.array of shape (n,d)
        Feature matrix
    Y : np.array of shape (n.k)
        One-hot encoded label matrix
    reg_lambda  : float
        Regularization hyper-parameter, >0

    Returns
    -------
    W_hat : np.array of shape (d,k)
        Weight matrix of ridge regression
    """
    W_hat = np.linalg.solve(X.T.dot(X) + reg_lambda*np.eye(X.shape[1]), X.T.dot(Y))
    return W_hat

def predict(W, X_prime):
    """
    Return ridge regression predictions

    Parameters
    ----------
```

```
    X : np.array of shape (m,d)
        Feature matrix
    W : np.array of shape (d,k)
        Weight matrix of ridge regression
    reg_lambda  : float
        Regularization hyper-parameter, >0

    Returns
    -------
    predictions : np.array of shape (m,)
        Ridge regression predictions
    """
    predictions = np.argmax(W.T.dot(X_prime.T), axis = 0)
    return predictions

if __name__ == "__main__":
    X_train, labels_train, X_test, labels_test = load_dataset()
    Y_train = one_hot_encoder(labels_train)
    W_hat = train(X_train, Y_train, 1e-4)
    labels_pred = predict(W_hat, X_test)
    labels_pred_train = predict(W_hat, X_train)
    print('Test error:', accuracy_error(labels_pred, labels_test))
    print('Train error:', accuracy_error(labels_pred_train, labels_train))
    #Test error: 0.14659999999999995
    #Train error: 0.14805000000000001
```

Code for A6.b

B.2 **Solution:**

a. See Figure 4. The code is below, after the figure.

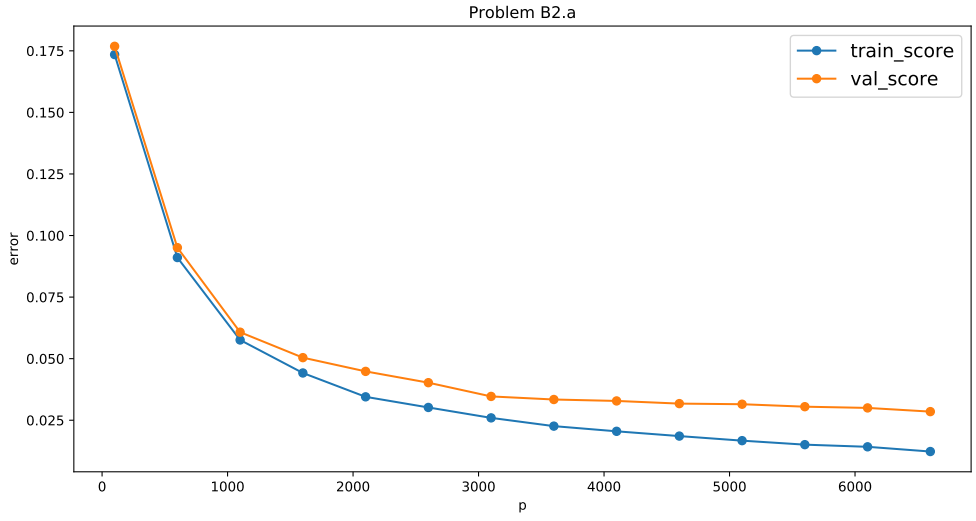

Figure 4: Problem B2.a

```
######################
#Problem B2.a, HW1
######################
import numpy as np
from mnist import MNIST
import matplotlib.pyplot as plt


def load_dataset():
    mndata = 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

def one_hot_encoder(labels_train):
    """
    Performs one-hot encoding of the labels

    Parameters
    ----------
    labels_train : np.array of shape (n,)
        Labels

    Returns
    -------
    np.array of shape (n,k)
        One-hot-encoded labels
    """
    return np.eye(len(set(labels_train)))[labels_train]

def accuracy_error(y_true, y_pred):
    """
    Trains ridge regression using closed-form solution

    Parameters
    ----------
    y_true : np.array of shape (m,)
        Vector of true labels
    y_pred : np.array of shape (m,)
        Vector of predicted labels

    Returns
    -------
    float
        error: 1-accuracy
    """
    return 1-np.mean(y_true == y_pred)

def train(X, Y, reg_lambda):
    """
    Trains ridge regression using closed-form solution

    Parameters
```

```
    ----------
    X : np.array of shape (n,d)
        Feature matrix
    Y : np.array of shape (n.k)
        One-hot encoded label matrix
    reg_lambda  : float
        Regularization hyper-parameter, >0

    Returns
    -------
    W_hat : np.array of shape (d,k)
        Weight matrix of ridge regression
    """
    W_hat = np.linalg.solve(X.T.dot(X) + reg_lambda*np.eye(X.shape[1]), X.T.dot(Y))
    return W_hat

def predict(W, X_prime):
    """
    Return ridge regression predictions

    Parameters
    ----------
    X : np.array of shape (m,d)
        Feature matrix
    W : np.array of shape (d,k)
        Weight matrix of ridge regression
    reg_lambda  : float
        Regularization hyper-parameter, >0

    Returns
    -------
    predictions : np.array of shape (m,)
        Ridge regression predictions
    """
    predictions = np.argmax(W.T.dot(X_prime.T), axis = 0)
    return predictions

def non_linear_transform(X, p, G=None, b=None):
    """
    Performs a non-linear transformation of the data: h(x) = cos(Gx+b)

    Parameters
    ----------
    X : np.array of shape (m,d)
        Feature matrix
    p : float
        Dimensionality parameter
    G : np.array of shape (p, d)
        Matrix for the affine projection of features
    b : np.array of shape (p,)
        Vector for the affine projection of features

    Returns
    -------
    np.array of shape (n,p)
```

```
        Transformed data
    G : np.array of shape (p, d)
        Matrix for the affine projection of features
    b : np.array of shape (p,)
        Vector for the affine projection of features

    """
    d = X.shape[1]
    if G is None:
        G = np.random.normal(0, 0.01, (p,d))
    if b is None:
        b = np.random.uniform(0, 2*np.pi, (p,))
    return np.cos(X.dot(G.T) + b), G, b

if __name__ == "__main__":
    X_train, labels_train, X_test, labels_test = load_dataset()
    Y_train = one_hot_encoder(labels_train)
    #create random permutation
    idx = np.random.permutation(X_train.shape[0])
    train_split = int(X_train.shape[0]*0.8)
    TRAIN = idx[:train_split]
    VAL = idx[train_split:]

    #create training set
    X_train80 = X_train[TRAIN, :]
    Y_train80 = Y_train[TRAIN] #one-hot encoded train labels
    labels_train80 = labels_train[TRAIN] #train labels

    #create validation set
    X_val = X_train[VAL, :]
    Y_val = Y_train[VAL]
    labels_val = labels_train[VAL]

    p_grid = np.arange(100, 5600, 500)
    val_score = []
    train_score = []
    for p in p_grid:
        print('Current p:', p)
        #Fit the model
        transformed_X_train80, G, b = non_linear_transform(X_train80, p)
        W_hat_p = train(transformed_X_train80, Y_train80, 1e-4)

        #Predict for train set:
        labels_train_pred = predict(W_hat_p, transformed_X_train80)
        train_score.append(accuracy_error(labels_train_pred, labels_train80))

        #Predict for validation set:
        transformed_X_val, _, _ = non_linear_transform(X_val, p, G, b)#use the same G,b
        labels_val_pred = predict(W_hat_p, transformed_X_val)
        val_score.append(accuracy_error(labels_val_pred, labels_val))


    plt.figure(figsize = (12, 6))
    plt.plot(p_grid, train_score, '-o', label = 'train_score')
    plt.plot(p_grid, val_score, '-o', label = 'val_score')
```

```
plt.title('Problem B2.a')
plt.xlabel('p')
plt.ylabel('error')
plt.legend(prop={'size': 14})
plt.savefig('b2_a.pdf')
plt.show()
```

Code for B2.a

b. First, we compute the test error using the best $\widehat{p}$ (looking at the plot in a., we chose $\widehat{p} = 3100$), we obtain (the code is below):

$$\boxed{\widehat{\epsilon}_{\text{test}}(\widehat{f}) = 0.0348.}$$

Now, X_i from the Hoeffding's inequality are iid indicator variables taking values in $\{0, 1\}$ corresponding to the i-th test example being classified correctly or not, respectively. So, to obtain the confidence interval, we plug in: $\delta = 0.05, m = 10000$ (test set size), $a = 0, b = 1$. Finally, we obtain $\sqrt{\frac{(b-a)^2 \log(2/\delta)}{2m}} \approx 0.01358102$ and

$$\mathbb{P}\left(\left|\widehat{\epsilon}_{\text{test}}(\widehat{f}) - \epsilon(\widehat{f})\right| \geq 0.01358102\right) \leq 0.05,$$

so the 95% CI for $\epsilon(\widehat{f})$ is $\boxed{0.0348 \pm 0.01358102}$ □. This question uses the same functions defined in the code for part a. and computes the test error as follows:

```
#B2.b
#Fit the model with the best p
#Based on the plot, p_hat = 3000
p_hat = 3100
transformed_X_train, G, b = non_linear_transform(X_train, p_hat)
W_hat_p = train(transformed_X_train, Y_train, 1e-4)

#Predict for validation set:
transformed_X_test, _, _ = non_linear_transform(X_test, p_hat, G, b)#use the same G,b
labels_test_pred = predict(W_hat_p, transformed_X_test)
print('Test error with p_hat:', accuracy_error(labels_test_pred, labels_test))
```
