Homework #1

CSE 446: Machine Learning
Eric Boris: 1976637
Collaborators: Sam Vanderlinda

Short Answer and "True or False" Conceptual questions

1. The answers to these questions should be answerable without referring to external materials. **Please input brief explanation for T/F questions as well**.

a. *[2 points]* In your own words, describe what bias and variance are. What is the bias-variance tradeoff?

Bias – The measure of how closely our model matches the best possible estimator, with lower values representing a closer match.

Variance – The measure of how much our model changes over i.i.d subsets of training data from the true probability distribution with lower values representing lower average differences.

Bias-Variance Tradeoff – Together, bias and variance represent the two types of learning error. However, one cannot usually be reduced completely without increasing the other. This is because **lowering bias tends to overfit the model which increases the variance**. Conversely, **lowering variance reduces fit which increases bias**. Thus, an optimal model balances bias and variance accepting some degree of error.

b. *[2 points]* What happens to bias and variance when model complexity increases/decreases?

Increased complexity leads to **decreased bias** and **increased variance**. **Decreased complexity** leads to **increased bias** and **decreased variance**.

c. *[1 points]* True or False: The bias of a model increases as the amount of available training data increases.

False – Typically more data allows learning better estimator parameters which reduce bias.

d. *[1 points]* True or False: The variance of a model decreases as the amount of available training data increases.

True – Typically more data reduces variance if the estimator is well chosen. However, if the estimator is poor, then this might not be the case.

e. *[1 points]* True or False: A learning algorithm will always generalize better if we use fewer features to represent our data.

False – Too few features can cause our model to generalize worse on unseen data.

f. *[2 points]* To obtain superior performance on new unseen data, should we use the training set or the test set to tune our hyperparameters?

Neither – We should use validation data to tune hyperparameters. However, if for some reason this is not an option, then we should use training data to tune hyperparameters. **Never** use test data to construct a model.

g. *[1 points]* True or False: The training error of a function on the training set provides an overestimate of the true error of that function.

False – Typically the model has lower training error on training data than the test error on test data since, although the model is tuned to training data with the assumption that test data is like training data, this isn't always the case and thus test error is likely higher than training error.

h. *[1 points]* True or False: Using L2 regularization when training a linear regression model encourages it to use less input features when making a prediction.

False – L2 Regularization reduces the impact of features on the model to reduce overfitting. It does not, however, actually remove or limit the number of features that the model has.

Maximum Likelihood Estimation (MLE)

---

2. Consider a model consisting of random variables X, Y and Z: $Y = Xw + Z$, where $Z \sim U[-0.5, 0.5]$. Assume that Z is a noise here (i.e. the independence holds) and $w \in \mathbb{R}$ is a fixed parameter (i.e. it is not random).

a. *[5 points]* Derive the probability density function (pdf) of Y conditioning on $\{X = x\}$.

Given that $Z \sim U[-0.5, 0.5]$, we write the PDF as:

$$P(Z = z) = \begin{cases} 1 & |z| \leq 0.5 \\ 0 & \text{otherwise} \end{cases}$$

Given that $Y = Xw + Z \implies Z = Y - Xw$, we write the PDF as:

$$P(Y = y|X = x) = \begin{cases} 1 & |y - xw| \leq 0.5 \\ 0 & \text{otherwise} \end{cases}$$

Thus

$$\boxed{Y|X \sim U[-0.5, 0.5]}$$

b. *[5 points]* Assume that you have n points of training data $\{(X_1, Y_1), (X_2, Y_2), \cdots, (X_n, Y_n)\}$ generated i.i.d in the above setting. Derive a maximum likelihood estimator of w for the conditional distribution of Y conditioning on $\{X = x\}$. Assume that $X_i > 0$ for all $i = 1, \ldots, n$ and note that MLE for this case may not be unique and you are required to report only one particular estimate.

Given that for all Y_i, X_i that $|Y_i - X_i w| \leq 0.5$ and that $X_i > 0$, w can be expressed as

$$\max\left(\frac{Y_i - 0.5}{X_i}\right) \leq w \leq \min\left(\frac{Y_i + 0.5}{X_i}\right)$$

Thus one MLE w can be expressed as

$$\boxed{w = \frac{1}{2}\left(\max\left(\frac{Y_i - 0.5}{X_i}\right) + \min\left(\frac{Y_i + 0.5}{X_i}\right)\right)}$$

---

3. You're a Reign FC fan, and the team is five games into its 2018 season. The number of goals scored by the team in each game so far are given below:

$$[2, 0, 1, 1, 2].$$

Let's call these scores $x_1, \ldots, x_5$. Based on your (assumed iid) data, you'd like to build a model to understand how many goals the Reign are likely to score in their next game. You decide to model the number of goals scored per game using a *Poisson distribution*. The Poisson distribution with parameter λ assigns every non-negative integer $x = 0, 1, 2, \ldots$ a probability given by

$$\text{Poi}(x|\lambda) = e^{-\lambda}\frac{\lambda^x}{x!}.$$

So, for example, if $\lambda = 1.5$, then the probability that the Reign score 2 goals in their next game is $e^{-1.5} \times \frac{1.5^2}{2!} \approx 0.25$. To check your understanding of the Poisson, make sure you have a sense of whether raising λ will mean more goals in general, or fewer.

a. *[5 points]* Derive an expression for the maximum-likelihood estimate of the parameter λ governing the Poisson distribution, in terms of your goal counts $x_1, \ldots, x_5$. (Hint: remember that the log of the likelihood has the same maximum as the likelihood function itself.)

Find the log likelihood function $l(x_i)$ for $1 \leq i \leq n$ of the Possion distribution.

$$\begin{aligned}
l(x_i) &= \ln\left(\prod_{i=1}^{n} \mathrm{e}^{-\lambda}\frac{\lambda^{x_i}}{x_i!}\right) \\
&= \sum_{i=1}^{n} \ln\left(\mathrm{e}^{-\lambda}\frac{\lambda^{x_i}}{x_i!}\right) \\
&= \sum_{i=1}^{n} \left(\ln\left(\mathrm{e}^{-\lambda}\right) + \ln\left(\lambda^{x_i}\right) - \ln\left(x_i!\right)\right) \\
&= \sum_{i=1}^{n} \left(-\lambda + x_i \ln(\lambda) - \ln(x_i!)\right) \\
&= -n\lambda + \ln(\lambda)\sum_{i=1}^{n} x_i - \sum_{i=1}^{n} x_i!
\end{aligned}$$

Take the derivative of the log likelihood function $l(x_i)$ with respect to λ.

$$\begin{aligned}
\frac{\mathrm{d}}{\mathrm{d}\lambda} l(x_i) &= \frac{\mathrm{d}}{\mathrm{d}\lambda}\left(-n\lambda + \ln(\lambda)\sum_{i=1}^{n} x_i - \sum_{i=1}^{n} x_i!\right) \\
&= -\frac{\mathrm{d}}{\mathrm{d}\lambda}(n\lambda) + \frac{\mathrm{d}}{\mathrm{d}\lambda}\left(\ln(\lambda)\sum_{i=1}^{n} x_i\right) - \frac{\mathrm{d}}{\mathrm{d}\lambda}\left(\sum_{i=1}^{n}\ln(x_i)\right) \\
&= -(n) + \left(\frac{1}{\lambda}\sum_{i=1}^{n} x_i\right) - (0) \\
&= -n + \frac{1}{\lambda}\sum_{i=1}^{n} x_i
\end{aligned}$$

Find the maximum likelihood estimator l_{MLE}.

$$\begin{aligned}
\frac{\mathrm{d}}{\mathrm{d}\lambda} l(x_i) &= 0 \\
-n + \frac{1}{\lambda}\sum_{i=1}^{n} x_i &= 0 \\
\frac{1}{\lambda}\sum_{i=1}^{n} x_i &= n \\
\frac{1}{\lambda} &= \frac{n}{\sum_{i=1}^{n} x_i} \\
\lambda &= \frac{1}{n}\sum_{i=1}^{n} x_i
\end{aligned}$$

Thus, when $n = 5$

$$\boxed{l_{\text{MLE}} = \frac{1}{5}\sum_{i=1}^{5} x_i}$$

b. *[5 points]* Suppose the team scores 4 goals in its sixth game. Derive the same expression for the estimate of the parameter λ as in the prior example, now using the 6 games $x_1, \ldots, x_5, x_6 = 4$.

When $n = 6$

$$\boxed{l_{\text{MLE}} = \frac{1}{6}\sum_{i=1}^{6} x_i}$$

c. *[5 points]* Given the goal counts, please give numerical estimates of λ after 5 and 6 games.

$l_{\text{MLE},5} = \frac{1}{5}\sum_{i=1}^{5} x_i = \frac{1}{5}\sum[2,0,1,1,2] = \boxed{\frac{6}{5}}$

$l_{\text{MLE},6} = \frac{1}{6}\sum_{i=1}^{6} x_i = \frac{1}{6}\sum[2,0,1,1,2,4] = \frac{10}{6} = \boxed{\frac{5}{3}}$

---

4. *[10 points]* In World War 2, the Allies attempted to estimate the total number of tanks the Germans had manufactured by looking at the serial numbers of the German tanks they had destroyed. The idea was that if there were n total tanks with serial numbers $\{1, \ldots, n\}$ then it is reasonable to expect the observed serial numbers of the destroyed tanks constituted a uniform random sample (without replacement) from this set. The exact maximum likelihood estimator for this so-called *German tank problem* is non-trivial and quite challenging to work out (try it!). For our homework, we will consider a much easier problem with a similar flavor.
Let $x_1, \ldots, x_n$ be independent, uniformly distributed on the continuous domain $[0, \theta]$ for some θ. What is the Maximum likelihood estimate for θ?

Find the likelihood function $l(\theta)$ of the Uniform distribution.

$$\begin{aligned} l(\theta) &= \prod_{i=1}^{n} \frac{1}{\theta}\,\mathbf{1}(x_i \in [0,\theta]) \\ &= \frac{1}{\theta^n}\,\mathbf{1}(x_i \in [0,\theta]) \end{aligned}$$

Note that for any maximum likelihood estimator θ_{MLE} that the maximum of $l(\theta_{\text{MLE}}) > 0$.

We know that $x_i \in [0,\theta]$ since otherwise $l(\theta) = 0$ by $\mathbf{1}(x_i \in [0,\theta])$.

Now, consider that since $\frac{1}{\theta^n}$ decreases as θ increases, that $\theta_{\text{MLE}} = \theta_{\text{min}}$.

Note that $0 < \theta$ so $0 < \theta_{\text{min}}$.

Since $x_i \in [0,\theta]$ then $x_i \leq \theta_{\text{min}}$.

We therefore conclude that $\theta_{\text{min}} = \max_i x_i$.

Thus

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

Overfitting

---

5. Suppose we have N labeled samples $S = \{(x_i, y_i)\}_{i=1}^{N}$ drawn i.i.d. from an underlying distribution $\mathcal{D}$. Suppose we decide to break this set into a set S_{train} of size N_{train} and a set S_{test} of size N_{test} samples for our training and test set, so $N = N_{\text{train}} + N_{\text{test}}$, and $S = S_{\text{train}} \cup S_{\text{test}}$. Recall the definition of the true least squares error of f:

$$\epsilon(f) = \mathbb{E}_{(x,y)\sim\mathcal{D}}[(f(x) - y)^2],$$

where the subscript $(x, y) \sim \mathcal{D}$ makes clear that our input-output pairs are sampled according to $\mathcal{D}$. Our training and test losses are defined as:

$$\widehat{\epsilon}_{\text{train}}(f) = \frac{1}{N_{\text{train}}} \sum_{(x,y)\in S_{\text{train}}} (f(x) - y)^2$$

$$\widehat{\epsilon}_{\text{test}}(f) = \frac{1}{N_{\text{test}}} \sum_{(x,y)\in S_{\text{test}}} (f(x) - y)^2$$

We then train our algorithm (for example, using linear least squares regression) using the training set to obtain $\widehat{f}$.

a. *[6 points]* (bias: the test error) Define $\mathbb{E}_{\text{train}}$ as the expectation over all training set S_{train} and $\mathbb{E}_{\text{test}}$ as the expectation over all testing set S_{test}. For all fixed f (before we've seen any data) show that

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

Use a similar line of reasoning to show that the test error is an unbiased estimate of our true error for $\hat{f}$. Specifically, show that:

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

- Show that $\mathbb{E}_{\text{train}}[\widehat{\epsilon}_{\text{train}}(f)] = \epsilon(f)$.

$$\begin{aligned}
\mathbb{E}_{\text{train}}[\widehat{\epsilon}_{\text{train}}(f)] &= \mathbb{E}_{\text{train}}\left[\frac{1}{N_{\text{train}}}\sum_{(x,\,y)\in S_{\text{train}}}(f(x)-y)^2\right] && \text{Given}\\
&= \frac{1}{N_{\text{train}}}\mathbb{E}_{\text{train}}\left[\sum_{(x,\,y)\in S_{\text{train}}}(f(x)-y)^2\right] && \text{By Linearity of Expectation}\\
&= \frac{1}{N_{\text{train}}}N_{\text{train}}\,\mathbb{E}_{(x,\,y)\sim\mathcal{D}}\left[(f(x)-y)^2\right] && \text{Since the points in } S_{\text{train}} \text{ are iid}\\
&= \mathbb{E}_{(x,y)\sim\mathcal{D}}\left[(f(x)-y)^2\right]\\
&= \epsilon(f)
\end{aligned}$$

- Show that $\mathbb{E}_{\text{test}}[\widehat{\epsilon}_{\text{test}}(f)] = \epsilon(f)$.

$$\begin{aligned}
\mathbb{E}_{\text{test}}[\widehat{\epsilon}_{\text{test}}(f)] &= \mathbb{E}_{\text{test}}\left[\frac{1}{N_{\text{test}}}\sum_{(x,\,y)\in S_{\text{test}}}(f(x)-y)^2\right] && \text{Given}\\
&= \frac{1}{N_{\text{test}}}\mathbb{E}_{\text{test}}\left[\sum_{(x,\,y)\in S_{\text{test}}}(f(x)-y)^2\right] && \text{By Linearity of Expectation}\\
&= \frac{1}{N_{\text{test}}}N_{\text{test}}\,\mathbb{E}_{(x,\,y)\sim\mathcal{D}}\left[(f(x)-y)^2\right] && \text{Since the points in } S_{\text{test}} \text{ are iid}\\
&= \mathbb{E}_{(x,y)\sim\mathcal{D}}\left[(f(x)-y)^2\right]\\
&= \epsilon(f)
\end{aligned}$$

- Since $\mathbb{E}_{\text{train}}[\widehat{\epsilon}_{\text{train}}(f)] = \epsilon(f)$ and $\mathbb{E}_{\text{test}}[\widehat{\epsilon}_{\text{test}}(f)] = \epsilon(f)$, thus

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

- Show that $\mathbb{E}_{\text{test}}[\widehat{\epsilon}_{\text{test}}(\widehat{f})] = \epsilon(\widehat{f})$.

$$\begin{aligned}
\mathbb{E}_{\text{test}}[\widehat{\epsilon}_{\text{test}}(\widehat{f})] &= \mathbb{E}_{\text{test}}\left[\frac{1}{N_{\text{test}}}\sum_{(x,\,y)\in S_{\text{test}}}\left(\widehat{f}(x)-y\right)^2\right] && \text{Given}\\
&= \frac{1}{N_{\text{test}}}\,\mathbb{E}_{\text{test}}\left[\sum_{(x,\,y)\in S_{\text{test}}}\left(\widehat{f}(x)-y\right)^2\right] && \text{By Linearity of Expectation}\\
&= \frac{1}{N_{\text{test}}}\,N_{\text{test}}\,\mathbb{E}_{(x,\,y)\sim\mathcal{D}}\left[\left(\widehat{f}(x)-y\right)^2\right] && \text{Since the points in } S_{\text{test}} \text{ are iid}\\
&= \mathbb{E}_{(x,y)\sim\mathcal{D}}\left[(\widehat{f}(x)-y)^2\right]\\
&= \epsilon(\widehat{f})
\end{aligned}$$

Thus,

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

b. *[5 points]* (bias: the train/dev error) Is the above equation true (in general) with regards to the training loss? Specifically, does $\mathbb{E}_{\text{train}}[\widehat{\epsilon}_{\text{train}}(\widehat{f})]$ equal $\mathbb{E}_{\text{train}}[\epsilon(\widehat{f})]$? If so, why? If not, give a clear argument as to where your previous argument breaks down.

In general **it is not true** that $\mathbb{E}_{\text{train}}[\widehat{\epsilon}_{\text{train}}(\widehat{f})] = \mathbb{E}_{\text{train}}[\epsilon(\widehat{f})]$. This is because, above, to go from $\frac{1}{N_{\text{test}}}\,\mathbb{E}_{\text{test}}\left[\sum_{(x,\,y)\in S_{\text{test}}}\left(\widehat{f}(x)-y\right)^2\right]$ to $\frac{1}{N_{\text{test}}}\,N_{\text{test}}\,\mathbb{E}_{(x,\,y)\sim\mathcal{D}}\left[\left(\widehat{f}(x)-y\right)^2\right]$ we had to know that x_i, y_i, and $\widehat{f}$ were iid from S_{train}. We knew this to be the case above because we're told that x_i and y_i are iid and that $\widehat{f}$ was found using only training data. However, since we don't always know how $\widehat{f}$ was found, we can't always assume that $\widehat{f}$ is iid from S_{train}, and thus it's not generally true that $\mathbb{E}_{\text{train}}[\widehat{\epsilon}_{\text{train}}(\widehat{f})] = \mathbb{E}_{\text{train}}[\epsilon(\widehat{f})]$.

c. *[8 points]* Let $\mathcal{F} = (f_1, f_2, \dots)$ be a collection of functions and let $\widehat{f}_{\text{train}}$ minimize the training error such that $\widehat{\epsilon}_{\text{train}}(\widehat{f}_{\text{train}}) \le \widehat{\epsilon}_{\text{train}}(f)$ for all $f \in \mathcal{F}$. Show that

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

(Hint: note that

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

where the second equality follows from the independence between the train and test set.)

Lemma: $\widehat{\epsilon}_{\text{train}}(\widehat{f}_{\text{train}}) \le \widehat{\epsilon}_{\text{train}}(f)\,\forall f\in\mathcal{F} \implies \mathbb{E}\left[\widehat{\epsilon}_{\text{train}}(\widehat{f}_{\text{train}})\right] \le \mathbb{E}\left[\widehat{\epsilon}_{\text{train}}(f)\right]\,\forall f\in\mathcal{F}$

$$
\begin{aligned}
\mathbb{E}_{\text{train}}\left[\widehat{\epsilon}_{\text{train}}(\widehat{f}_{\text{train}})\right] &= \mathbb{E}_{\text{train}}\left[\widehat{\epsilon}_{\text{train}}(\widehat{f}_{\text{train}})\right] * 1 && \text{Trivially} \\
&= \mathbb{E}_{\text{train}}\left[\widehat{\epsilon}_{\text{train}}(\widehat{f}_{\text{train}})\right] \sum_{f\in\mathcal{F}} \mathbb{P}\left(\widehat{f}_{\text{train}} = f\right) && \text{Sum of PMF} = 1 \\
&= \sum_{f\in\mathcal{F}} \mathbb{E}_{\text{train}}\left[\widehat{\epsilon}_{\text{train}}(\widehat{f}_{\text{train}})\right] \mathbb{P}\left(\widehat{f}_{\text{train}} = f\right) && \text{By Linearity of Expectation} \\
&\le \sum_{f\in\mathcal{F}} \mathbb{E}_{\text{train}}\left[\widehat{\epsilon}_{\text{train}}(f)\right] \mathbb{P}\left(\widehat{f}_{\text{train}} = f\right) && \text{From above lemma} \\
&= \sum_{f\in\mathcal{F}} \mathbb{E}_{\text{test}}\left[\widehat{\epsilon}_{\text{test}}(f)\right] \mathbb{P}\left(\widehat{f}_{\text{train}} = f\right) && \text{Proven in a.} \\
&= \sum_{f\in\mathcal{F}} \mathbb{E}_{\text{test}}\left[\widehat{\epsilon}_{\text{test}}(f)\right] \mathbb{E}_{\text{train}}\left[\mathbf{1}\{\widehat{f}_{\text{train}} = f\}\right] && \text{Given in hint} \\
&= \sum_{f\in\mathcal{F}} \mathbb{E}_{\text{train,test}}\left[\widehat{\epsilon}_{\text{test}}(f)\, \mathbf{1}\{\widehat{f}_{\text{train}} = f\}\right] && \text{Given in hint} \\
&= \mathbb{E}_{\text{train,test}}\left[\widehat{\epsilon}_{\text{test}}(\widehat{f}_{\text{train}})\right] && \text{Given in hint}
\end{aligned}
$$

Thus,

$$\boxed{\mathbb{E}_{\text{train}}\left[\widehat{\epsilon}_{\text{train}}(\widehat{f}_{\text{train}})\right] \le \mathbb{E}_{\text{train,test}}\left[\widehat{\epsilon}_{\text{test}}(\widehat{f}_{\text{train}})\right]}$$

Polynomial Regression

Relevant Files[1]

- **polyreg.py**
- `linreg_closedform.py`
- `test_polyreg_univariate.py`
- `test_polyreg_learningCurve.py`
- `data/polydata.dat`

Recall that polynomial regression learns a function $h_{\boldsymbol{\theta}}(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \ldots + \theta_d x^d$. In this case, d represents the polynomial's degree. We can equivalently write this in the form of a linear model

$$h_{\boldsymbol{\theta}}(x) = \theta_0\phi_0(x) + \theta_1\phi_1(x) + \theta_2\phi_2(x) + \ldots + \theta_d\phi_d(x) \ , \tag{1}$$

using the basis expansion that $\phi_j(x) = x^j$. Notice that, with this basis expansion, we obtain a linear model where the features are various powers of the single univariate x. We're still solving a linear regression problem, but are fitting a polynomial function of the input.

Implement regularized polynomial regression in `polyreg.py`. You may implement it however you like, using gradient descent or a closed-form solution. However, I would recommend the closed-form solution since the data sets are small; for this reason, we've included an example closed-form implementation of linear regression in `linreg_closedform.py` (you are welcome to build upon this implementation, but make CERTAIN you understand it, since you'll need to change several lines of it). You are also welcome to build upon your implementation from the previous assignment, but you must follow the API below. Note that all matrices are actually 2D numpy arrays in the implementation.

- `__init__(degree=1, regLambda=1E-8)` : constructor with arguments of d and λ
- `fit(X,Y)`: method to train the polynomial regression model
- `predict(X)`: method to use the trained polynomial regression model for prediction

[1] **Bold text** indicates files or functions that you will need to complete; you should not need to modify any of the other files.

- `polyfeatures(X, degree)`: expands the given $n \times 1$ matrix X into an $n \times d$ matrix of polynomial features of degree d. Note that the returned matrix will not include the zero-th power.

Note that the `polyfeatures(X, degree)` function maps the original univariate data into its higher order powers. Specifically, X will be an $n \times 1$ matrix ($X \in \mathbb{R}^{n \times 1}$) and this function will return the polynomial expansion of this data, a $n \times d$ matrix. Note that this function will **not** add in the zero-th order feature (i.e., $x_0 = 1$). You should add the x_0 feature separately, outside of this function, before training the model. By not including the x_0 column in the matrix `polyfeatures()`, this allows the `polyfeatures` function to be more general, so it could be applied to multi-variate data as well. (If it did add the x_0 feature, we'd end up with multiple columns of 1's for multivariate data.)

Also, notice that the resulting features will be badly scaled if we use them in raw form. For example, with a polynomial of degree $d = 8$ and $x = 20$, the basis expansion yields $x^1 = 20$ while $x^8 = 2.56 \times 10^{10}$ – an absolutely huge difference in range. Consequently, we will need to standardize the data before solving linear regression. Standardize the data in `fit()` after you perform the polynomial feature expansion. You'll need to apply the same standardization transformation in `predict()` before you apply it to new data.

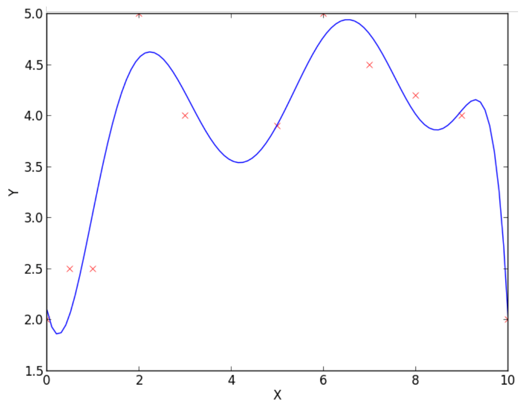
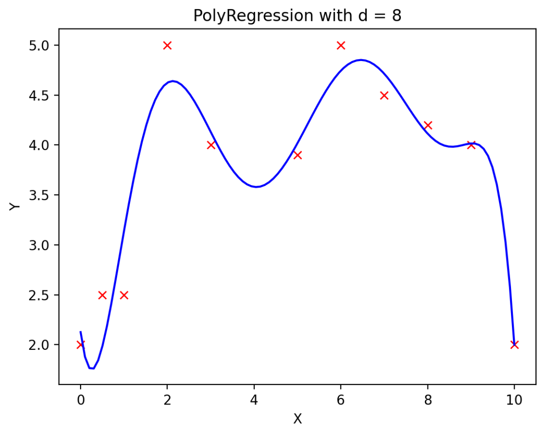


Figure 1: Fit of polynomial regression with $\lambda = 0$ and $d = 8$

Run `test_polyreg_univariate.py` to test your implementation, which will plot the learned function. In this case, the script fits a polynomial of degree $d = 8$ with no regularization $\lambda = 0$. From the plot, we see that the function fits the data well, but will not generalize well to new data points. Try increasing the amount of regularization, and examine the resulting effect on the function.

Results of training Polynomial Regression model of degree d = 8.

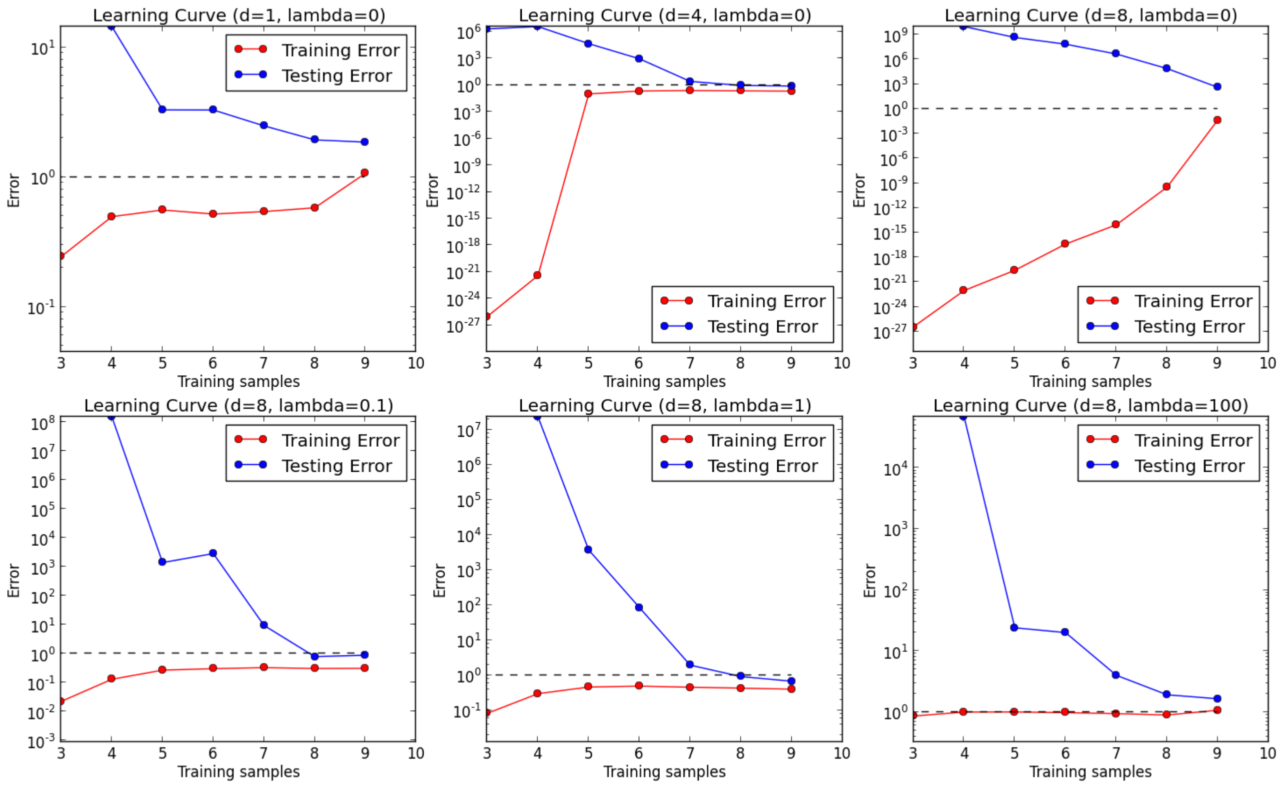


7. *[15 points]* In this problem we will examine the bias-variance tradeoff through learning curves. Learning curves provide a valuable mechanism for evaluating the bias-variance tradeoff. Implement the `learningCurve()` function in `polyreg.py` to compute the learning curves for a given training/test set. The `learningCurve(Xtrain, ytrain, Xtest, ytest, degree, regLambda)` function should take in the training data (`Xtrain`, `ytrain`), the testing data (`Xtest`, `ytest`), and values for the polynomial degree d and regularization parameter λ.
The function should return two arrays, `errorTrain` (the array of training errors) and `errorTest` (the array of testing errors). The i^{th} index (start from 0) of each array should return the training error (or testing error) for learning with $i+1$ training instances. Note that the 0^{th} index actually won't matter, since we typically start displaying the learning curves with two or more instances.

When computing the learning curves, you should learn on `Xtrain`$[0{:}i]$ for $i = 1, \ldots, \text{numInstances}(\texttt{Xtrain}) + 1$, each time computing the testing error over the **entire** test set. There is no need to shuffle the training data, or to average the error over multiple trials – just produce the learning curves for the given training/testing sets with the instances in their given order. Recall that the error for regression problems is given by

$$\frac{1}{n}\sum_{i=1}^{n}(h_{\boldsymbol{\theta}}(\mathbf{x}_i) - y_i)^2 \quad . \tag{2}$$

Once the function is written to compute the learning curves, run the `test_polyreg_learningCurve.py` script to plot the learning curves for various values of λ and d. You should see plots similar to the following:



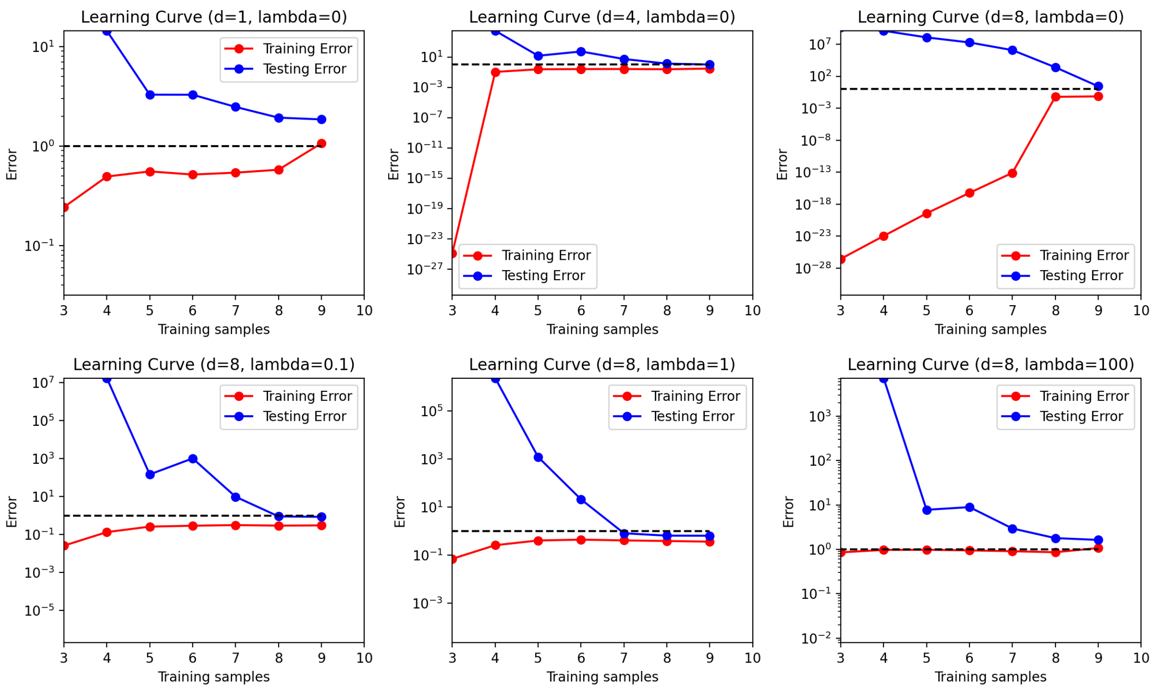
Notice the following:

- The y-axis is using a log-scale and the ranges of the y-scale are all different for the plots. The dashed black line indicates the $y = 1$ line as a point of reference between the plots.
- The plot of the unregularized model with $d = 1$ shows poor training error, indicating a high bias (i.e., it is a standard univariate linear regression fit).

- The plot of the unregularized model ($\lambda = 0$) with $d = 8$ shows that the training error is low, but that the testing error is high. There is a huge gap between the training and testing errors caused by the model overfitting the training data, indicating a high variance problem.
- As the regularization parameter increases (e.g., $\lambda = 1$) with $d = 8$, we see that the gap between the training and testing error narrows, with both the training and testing errors converging to a low value. We can see that the model fits the data well and generalizes well, and therefore does not have either a high bias or a high variance problem. Effectively, it has a good tradeoff between bias and variance.
- Once the regularization parameter is too high ($\lambda = 100$), we see that the training and testing errors are once again high, indicating a poor fit. Effectively, there is too much regularization, resulting in high bias.

Please include both your code and the generated plots in your homework. Make absolutely certain that you understand these observations, and how they relate to the learning curve plots. In practice, we can choose the value for λ via cross-validation to achieve the best bias-variance tradeoff.

Results of training and testing error over multiple degrees d and lambdas λ.



```
'''
    Template for polynomial regression
   AUTHOR Eric Eaton, Xiaoxiang Hu
'''

import numpy as np
from numpy.linalg import pinv
from copy import deepcopy

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

class PolynomialRegression:

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

  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
```

```
        """
        # Prevent side-effects.
        X_copy = deepcopy(X)

        # Expand X to degree d.
        for d in range(1, degree):
            X_copy = np.concatenate((X_copy, X ** d), axis=1)

        return X_copy

    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
        """
        # Validate input.
        assert (X.shape == y.shape)

        X = self.preprocess(X, train=True)

        # Compute Theta using closed form solution with regularized matrix M = \lambda
            * I.
        # \Theta = (X^T X + M)^{-1} X^T y
        M = self.reg_lambda * np.eye(X.shape[1])
        M[0, 0] = 0  # Ignore offset.
        self.theta = np.dot(np.dot(pinv(np.dot(X.T, X) + M), X.T), 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

        """
        X = self.preprocess(X)

        # Return y_hat.
        return np.dot(X, self.theta)

    def preprocess(self, X, train=False):
        """
        Prepare the given array for training and predicting.
        Arguments:
            X is a n-by-1 numpy array.
            train is a flag representing whether model is training or predicting.
        Returns:
            An n-by-d numpy array, standardized with an offset column.
        """
        # Prevent side-effects.
        X = deepcopy(X)

        # Let n be the number of features in X.
        n = X.shape[0]

        # Expand to polynomial degree d.
        X = self.polyfeatures(X, self.degree)

        # Only compute mean and std while training.
        # Otherwise, use the computed mean and std during prediction.
        if train:
```

```
      self.mean = np.mean(X, axis=0) if n > 1 else 0
      self.std = np.std(X, axis=0) if n > 1 else 1

    # Standardize using Z-score.
    X = (X - self.mean) / self.std

    # Add offset column.
    X = np.concatenate((np.ones([n, 1]), X), axis=1)

    return X

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

def error(y_hat, y):
  """
  Return the square error.
  Arguments:
   y_hat -- the predicted output.
   y    -- the actual output.
  Returns:
   The square error.
  """
  return np.mean((y_hat - y) ** 2)

def learningCurve(X_train, Y_train, X_test, Y_test, reg_lambda, degree):
  """
  Compute learning curve
  Arguments:
   X_train -- Training X, n-by-1 matrix
   Y_train -- Training y, n-by-1 matrix
   X_test -- Testing X, m-by-1 matrix
   Y_test -- Testing Y, m-by-1 matrix
   regLambda -- regularization factor
   degree -- polynomial degree
  Returns:
   error_train -- error_train[i] is the training accuracy using
   model trained by X_train[0:(i+1)]
   error_test -- error_train[i] is the testing accuracy using
   model trained by X_train[0:(i+1)]
  Note:
   error_train[0:1] and error_test[0:1] won't actually matter, since we start
       displaying the learning curve at n = 2 (or higher)
  """
  n = len(X_train)

  # Let the following store model errors
  # s.t. error_t..[i] = error on model_n where n = i+1.
  error_train = np.zeros(n)
  error_test = np.zeros(n)

  # Compute model performance on differently sized datasets
  # starting at n=2 and increasing.
  for i in range(1, n):
    # Get the training sets.
    X = X_train[0 : i + 1]
    y = Y_train[0 : i + 1]

    # Get and train the model.
    model = PolynomialRegression(degree, reg_lambda)
    model.fit(X, y)

    # Get the model's predictions on training and test sets.
    y_hat_train = model.predict(X)
    y_hat_test = model.predict(X_test)

    # Compute and store model performance.
    error_train[i] = error(y_hat_train, y)
```

```
        error_test[i] = error(y_hat_test, Y_test)

    return error_train, error_test
```

Ridge Regression on MNIST

8. In this problem we will implement a regularized least squares classifier for the MNIST data set. The task is to classify handwritten images of numbers between 0 to 9.

You are **NOT** allowed to use any of the prebuilt classifiers in `sklearn`. Feel free to use any method from `numpy` or `scipy`. Remember: if you are inverting a matrix in your code, you are probably doing something wrong (Hint: look at `scipy.linalg.solve`).

Get the data from `https://pypi.python.org/pypi/python-mnist`.
Load the data as follows:

```
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
```

Each example has features $x_i \in \mathbb{R}^d$ (with $d = 28*28 = 784$) and label $z_j \in \{0, \ldots, 9\}$. You can visualize a single example x_i with `imshow` after reshaping it to its original 28×28 image shape (and noting that the label z_j is accurate). We wish to learn a predictor $\widehat{f}$ that takes as input a vector in $\mathbb{R}^d$ and outputs an index in $\{0, \ldots, 9\}$. We define our training and testing classification error on a predictor f as

$$\widehat{\epsilon}_{\text{train}}(f) = \frac{1}{N_{\text{train}}} \sum_{(x,z)\in \text{Training Set}} \mathbf{1}\{f(x) \neq z\}$$

$$\widehat{\epsilon}_{\text{test}}(f) = \frac{1}{N_{\text{test}}} \sum_{(x,z)\in \text{Test Set}} \mathbf{1}\{f(x) \neq z\}$$

We will use one-hot encoding of the labels, i.e. of (x, z) the original label $z \in \{0, \ldots, 9\}$ is mapped to the standard basis vector e_z where e_z is a vector of all zeros except for a 1 in the zth position. We adopt the notation where we have n data points in our training objective with features $x_i \in \mathbb{R}^d$ and label one-hot encoded as $y_i \in \{0, 1\}^k$ where in this case $k = 10$ since there are 10 digits.

a. *[10 points]* In this problem we will choose a linear classifier to minimize the regularized least squares objective:

$$\widehat{W} = \text{argmin}_{W\in\mathbb{R}^{d\times k}} \sum_{i=0}^{n} \|W^T x_i - y_i\|_2^2 + \lambda \|W\|_F^2$$

Note that $\|W\|_F$ corresponds to the Frobenius norm of W, i.e. $\|W\|_F^2 = \sum_{i=1}^{d} \sum_{j=1}^{k} W_{i,j}^2$. To classify a point x_i we will use the rule $\arg\max_{j=0,\dots,9} e_j^T \widehat{W}^T x_i$. Note that if $W = \begin{bmatrix} w_1 & \dots & w_k \end{bmatrix}$ then

$$\sum_{i=0}^{n} \|W^T x_i - y_i\|_2^2 + \lambda \|W\|_F^2 = \sum_{j=0}^{k} \left[\sum_{i=1}^{n} (e_j^T W^T x_i - e_j^T y_i)^2 + \lambda \|W e_j\|^2 \right]$$
$$= \sum_{j=0}^{k} \left[\sum_{i=1}^{n} (w_j^T x_i - e_j^T y_i)^2 + \lambda \|w_j\|^2 \right]$$
$$= \sum_{j=0}^{k} \left[\|X w_j - Y e_j\|^2 + \lambda \|w_j\|^2 \right]$$

where $X = \begin{bmatrix} x_1 & \dots & x_n \end{bmatrix}^\top \in \mathbb{R}^{n\times d}$ and $Y = \begin{bmatrix} y_1 & \dots & y_n \end{bmatrix}^\top \in \mathbb{R}^{n\times k}$. Show that

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

Show that $\widehat{W} = \left(X^T X + \lambda I\right)^{-1} X^T Y$.

$$\widehat{W} = \operatorname{argmin}_{W\in\mathbb{R}^{d\times k}} \sum_{i=0}^{n} \|W^T x_i - y_i\|_2^2 + \lambda \|W\|_F^2 \qquad \text{Given}$$
$$= \operatorname{argmin}_{W\in\mathbb{R}^{d\times k}} \sum_{j=0}^{n} \left[\|X w_j - Y e_j\|^2 + \lambda \|w_j\|^2 \right] \qquad \text{Given in hint}$$
$$= \operatorname{argmin}_{W\in\mathbb{R}^{d\times k}} \sum_{j=0}^{n} \left[(X w_j - Y e_j)^T (X w_j - Y e_j) + \lambda (w_j)^T (w_j) \right]$$

Lemma 1: $X^T X$ is positive since $z^T X^T X z = (Xz)^T (Xx) = \|Xz\|^2 \geq 0$ for all $z \in \mathbb{R}^n$.
Lemma 2: λI is positive since $z^T \lambda I z = \lambda z^T z = \lambda \|z\|^2 > 0$ for all $z \in \mathbb{R}^n \setminus 0$ and $\lambda > 0$.
Lemma 3: $(X^T X + \lambda I)$ is invertible because by Lemma 1 and Lemma 2, it is postive.
Find the argmin by taking the derivative and setting equal to zero.

$$\frac{\partial}{\partial w_j} \sum_{j=0}^{n} \left[(X w_j - Y e_j)^T (X w_j - Y e_j) + \lambda (w_j)^T (w_j) \right] = 0$$
$$2X^T (X w_j - Y e_j) + \lambda 2 w_j = 0$$
$$(X^T X + \lambda I)^{-1} X^T Y e_j = wj$$

Thus

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

b. *[10 points]*

- Code up a function `train` that takes as input $X \in \mathbb{R}^{n\times d}$, $Y \in \{0,1\}^{n\times k}$, $\lambda > 0$ and returns $\widehat{W}$.
- Code up a function `predict` that takes as input $W \in \mathbb{R}^{d\times k}$, $X' \in \mathbb{R}^{m\times d}$ and returns an m-length vector with the ith entry equal to $\arg\max_{j=0,\dots,9} e_j^T W^T x_i'$ where x_i' is a column vector representing the ith example from X'.
- Train $\widehat{W}$ on the MNIST training data with $\lambda = 10^{-4}$ and make label predictions on the test data. **What is the training and testing error?** Note that they should both be about 15%.

Training and Testing Error on Ridge Regression model.

Training Error = 0.1481

Testing Error = 0.1466

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

def load_dataset():
  '''
  Load the mnist handwritten digits dataset.
  Returns:
   X_train  -- The training dataset inputs.
   labels_train  -- The training dataset labels.
   X_test  -- The testing dataset inputs.
   labels_test  -- The testing dataset labels.
  '''
  mndata = MNIST('./data/python-mnist/data/')
  X_train, labels_train = map(np.array, mndata.load_training())
  X_test, labels_test = map(np.array, mndata.load_testing())
  X_train = X_train / 255.0
  X_test = X_test / 255.0
  return X_train, labels_train, X_test, labels_test

def one_hot(array):
  '''
  Returns a one-hot encoding of the given array.
  Arguments:
   array -- (n, ) numpy array.
  Returns:
   array'  -- (n, k) numpy array one hot encoding.
  '''
  n = len(set(array))
  array_prime = np.eye(n)[array]
  return array_prime

def error(y, y_hat):
  '''
  Compute the error between the two sets of labels.
  Arguments:
   y  -- (n, ) numpy array of acutal labels.
   y_hat  -- (n, ) numpy array of predicted labels.
  Returns:
   error  -- The measure of error between the two sets.
  '''
  error = 1 - np.mean(y == y_hat)
  return error

class RidgeRegression:
  def train(self, X, y, reg_lambda=1E-4):
    '''
    Trains the ridge regression model using a closed-form solution.
    Arguments:
     X   -- (n, m) numpy array of training features.
     y    -- (n, k) numpy array of training labels.
     reg_lambda  -- The regularization parameter.
    Returns:
     A ridge regression trained model.
    '''
    # Let M be the regularized matrix s.t. M = \lambda * I
    M = reg_lambda * np.eye(X.shape[1])

    # Let W_hat be the trained weights s.t. W_hat = (X^T X + M)^{-1} X^T y
    W_hat = np.linalg.solve(np.dot(X.T, X) + M, np.dot(X.T, y))

    return W_hat

  def predict(self, X, W):
    '''
    Predicts the labels of the given dataset using the given model.
    Arguments:
     X  -- (n, m) numpy array of features.
     W    -- (m, k) numpy array of weights.
    Returns:
     y_hat -- (n, ) numpy array of predicted labels.
    '''
    y_hat = np.argmax(np.dot(W.T, X.T), axis=0)
    return y_hat


if __name__ == '__main__':
  # Load the data.
  print('Loading data')
  X_train, labels_train, X_test, labels_test = load_dataset()

  # Use one-hot encoding for training the model.
  y_train = one_hot(labels_train)

   # Train the ridge regression model.
  print('Training model')
  model = RidgeRegression()
  W_hat = model.train(X_train, y_train)

  # Predict using the model.
  print('Predicting')
  y_hat_train = model.predict(X_train, W_hat)
  y_hat_test = model.predict(X_test, W_hat)

  # Compute the model's error.
  print('Computing error')
  train_error = error(labels_train, y_hat_train)
  test_error = error(labels_test, y_hat_test)

  # Output the results.
  print(f'Training error: {train_error}')
  print(f'Testing error: {test_error}')
```