ASSIGNMENT 2: [IFT6390]

LÉA RICARD & JOSEPH D. VIVIANO

1. Linear and non-linear regularized regression

1.1. **Linear Regression.** *Consider a regression problem for which we have a training dataset $D_n = (\mathbf{x}^1, t^1), ..., (\mathbf{x}^n, t^n)$, a linear transformation defined as $f(x) = \mathbf{w}^T\mathbf{x} + b$, with $\mathbf{x}^{(i)} \in \mathbb{R}^d$, and $t^{(i)} \in \mathbb{R}$.*

1.1.1. *Precise this model's set of parameters θ, as well as the nature and dimensionality of each of them.*

The vector $\boldsymbol{w}$ contains the parameters of this model, as well as the bias term which is a scalar, so $\theta = \{w_1, w_2, ..., w_d, b\}$. $\theta \in \mathbb{R}^{d+1}$.

1.1.2. *The loss function typically used for linear regression is the quadratic loss:*

$$L\big((x,t), f\big) = \big(f(x) - t\big)^2 \tag{1.1}$$

Give the precise mathematical formula of the empirical risk $\hat{R}$ on the set D_n as the sum of the losses on this set.

We can find the empirical risk by rewriting the negative log likelihood (NLL) formula (equation 1.2), where $p(t^{(i)}|\mathbf{x}^{(i)}, \theta)$ is following a Gaussian distribution.

$$NLL(\theta) = -\sum_{i=1}^{N} \log p(t^i|\mathbf{x}_i, \theta) \tag{1.2}$$

$$NLL(\theta) = -\sum_{i=1}^{N} \log[(\frac{1}{2\pi\sigma^2})^{1/2} \exp(\frac{1}{2\sigma^2}(t^i - \mathbf{w}^T\mathbf{x}_i)^2] \tag{1.3}$$

$$= \frac{1}{2\sigma^2}\sum_{i=1}^{N}(t^i - \mathbf{w}^T\mathbf{x}_i)^2 - \frac{N}{2}\log(2\pi\sigma^2) \tag{1.4}$$

Where,

$$\hat{R}(\theta) = \sum_{i=1}^{N}(t^i - \mathbf{w}^T\mathbf{x}_i)^2 \tag{1.5}$$

is the residual sum of squares (RRS), or quadratic loss. On a dataset of size N, the empirical risk is given by this equation. Note that b is included in the vector $\mathbf{w^T}$

Date: Nov 2018.

and there is a corresponding dummy column of ones in $\mathbf{x}$. For the remainder of the assignment, we will use this notation.

1.1.3. *Following the principle of Empirical Risk Minimization (ERM), we are going to seek the parameters which yield the smallest quadratic loss. Write a mathematical formulation of this minimization problem.*

To estimate the parameters, we minimize the negative log likelihood (NNL).

$$\hat{\theta} = argmin - \sum_{i=1}^{N} \log p(t^i | \mathbf{x}_i, \theta) \tag{1.6}$$

$$= argminNLL(\theta) \tag{1.7}$$

set$_x scale('log')$
From equation 1.4, we can see that only the first part (RSS) is dependent on $\mathbf{w}$. Hence, to minimize NLL, we only have to minimize the residual sum of squares (RSS).

Therefore we can write the minimization objective as:

$$\hat{\theta} = argmin \sum_{i=1}^{N} (t^i - \mathbf{w}^T \mathbf{x}_i)^2 \tag{1.8}$$

We can rewrite the objective function in another form, which will be more adapted for differentiation using the properties of matrix multiplication:

$$\hat{R}(\theta) = \frac{1}{2}(\mathbf{t} - \mathbf{X}\mathbf{w})^T(\mathbf{t} - \mathbf{X}\mathbf{w}) \tag{1.9}$$

$$= \frac{1}{2}\mathbf{w}^T(\mathbf{X}^T\mathbf{X})\mathbf{w} - \mathbf{w}^T(\mathbf{X}^T\mathbf{t}) \tag{1.10}$$

Where $\mathbf{X}$ is the vector of all $\mathbf{x_i}$, which allows us to represent the summation over all $\mathbf{x_i}$ via matrix multiplication. Therefore, follows from the previous formulation because $X^T\mathbf{y} = \sum_{i=1}^{N} \mathbf{x_i} y_i$ and $\mathbf{X}\mathbf{X}^T = \sum_{i=1}^{N} \mathbf{x_i}\mathbf{x_i}^T$.

Using the following proprieties (where, b and a are vectors and A a matrix):

$$\frac{\partial a^T A a}{\partial a} = (A + A^T)a \tag{1.11}$$

$$\frac{\partial b^T a}{\partial a} = b \tag{1.12}$$

$$b^T A b = tr(b^T A b) = tr(b b^T A) = tr(A b b^T) \tag{1.13}$$

We can define the gradient over the NLL as follows:

$$\nabla(\hat{R}(\theta)) = [\mathbf{X}^T\mathbf{X}\mathbf{w} - \mathbf{X}^T\mathbf{t}] \tag{1.14}$$

$$= \sum_{i=1}^{N} \mathbf{x}_i(\mathbf{X}^T\mathbf{x}_i - t^i) \tag{1.15}$$

To find the normal equation, we put the gradient found earlier equal to zero and isolate $\hat{\mathbf{w}}$

$$[\mathbf{X}^T\mathbf{X}\mathbf{w} - \mathbf{X}^T\mathbf{t}] = 0 \tag{1.16}$$

$$\mathbf{X}^T\mathbf{X}\mathbf{w} = \mathbf{X}^T\mathbf{t} \tag{1.17}$$

$$\hat{\mathbf{w}} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{t} \tag{1.18}$$

1.1.4. *A general algorithm for solving this optimization problem is gradient descent. Give a formula for the gradient of the empirical risk with respect to each parameter.*

The formula of the gradient of the empirical risk is:

$$\nabla(\hat{R}(\theta) = [\mathbf{X}^T\mathbf{X}\mathbf{w} - \mathbf{X}^T\mathbf{t}] \tag{1.19}$$

Using the gradient descent method, for each step, we update the w parameter by substracting the gradient multiplied by a step size parameter (regularized by a lambda factor) until the gradient is close enough to zero.

$$\theta \leftarrow \theta - \lambda(\nabla(\hat{R}(\theta)) \tag{1.20}$$

1.1.5. *Define the error of the model on a single point (x, t) by f(x) - t. Explain in English the relationship between the empirical risk gradient and the errors on the training set.*

The greater the error on the training (as measured by $RSS(\theta)$), the larger the slope of the empirical risk gradient. The direction of the gradient can be used to find a better set of parameters θ that will give a smaller error. In the case of *argmin*, we move along the negative gradient.

2. Ridge Regression

Instead of $\hat{R}$, we will now consider a regularized empirical risk: $\widetilde{R}\hat{+}\lambda L(\theta)$. Here L takes the parameters θ and returns a scalar penalty. This penalty is smaller for parameters for which we have an a priori preference. The scalar $\lambda \geq 0$ is an hyperparameter that controls how much we favor minimizing the empirical risk versus this penalty. Note that we find the unregularized empirical risk when $\lambda = 0$. We will consider a regularization called Ridge, or weight decay that penalizes the squared norm (l^2 norm) of the weights (but not the bias): $L(\theta) = ||w||^2 = \sum_{k=1}^{d} \mathbf{w}_k^2$. We want to minimize $\widetilde{R}$ rather than $\hat{R}$.

2.1. *Express the gradient of $\widetilde{R}$. How does it differ from the unregularized empirical risk gradient?*

The gradient of the loss term and regularization term are independent during differentiation, so we simply get:

$$\begin{aligned} \nabla(\hat{R}(\theta) + \lambda||\mathbf{w}||_2^2) &= \nabla[\hat{R}(\theta)] + \lambda\nabla[\mathbf{w}^T\mathbf{w}] && (2.1)\\ &= [\mathbf{X}^T\mathbf{X}\mathbf{w} - \mathbf{X}^T\mathbf{t}] + \lambda[2\mathbf{w}] && (2.2) \end{aligned}$$

2.2. Gradient descent is defined as:

$$\theta \leftarrow \theta - \eta\frac{\partial J}{\partial\theta}(\theta) \tag{2.3}$$

$$\theta \leftarrow \theta - \eta([\mathbf{X}^T\mathbf{X}\mathbf{w} - \mathbf{X}^T\mathbf{t}] + \lambda[2\mathbf{w}]) \tag{2.4}$$

We use this framework and the results from the previous question to produce:

Listing 1. A Python code

```
step_size = 0.2 # hyper-parameter
threshold = 10e-4 # hyper-parameter
lambda = 4 # hyper-parameter
d_Jtheta = uniform(d) # vector randomly initialized to small values
                      # distributed around zero, length d
while sum(abs(d_Jtheta)) > threshold:
    # t is our targets
    # X is our predictors
    # w is our weight matrix

    d_Jtheta = X.T.dot(X).dot(w) - X.T.dot(t) + lambda * 2 * w
    theta -= step_size * d_Jtheta
```

2.3. *Define a matrix formulation for the empirical risk and it's gradient.*

The analytical solution to this problem is defined by the normal equation. To find this, we start with the empirical risk $\hat{R}$ and it's gradient, previously defined for linear regression. Let's start with empirical risk:

$$\hat{R}(\theta) = \frac{1}{2}\mathbf{w}^T(\mathbf{X}^T\mathbf{X})\mathbf{w} - \mathbf{w}^T(\mathbf{X}^T\mathbf{t}) + \lambda\mathbf{w}^T\mathbf{w}$$

$$\begin{aligned}\hat{R}(\theta) = \frac{1}{2}\begin{bmatrix} w^{(1)} & \dots & w^{(d)} \end{bmatrix} \Bigg(\begin{bmatrix} x_1^{(1)} & \dots & x_1^{(d)} \\ \vdots & \ddots & \vdots \\ x_d^{(1)} & \dots & x_d^{(n)} \end{bmatrix} \begin{bmatrix} x_1^{(1)} & \dots & x_d^{(1)} \\ \vdots & \ddots & \vdots \\ x_1^{(d)} & \dots & x_d^{(n)} \end{bmatrix} \Bigg) \begin{bmatrix} w^{(1)} \\ \vdots \\ w^{(d)} \end{bmatrix} \\ - \begin{bmatrix} w^{(1)} & \dots & w^{(d)} \end{bmatrix} \Bigg(\begin{bmatrix} x_1^{(1)} & \dots & x_1^{(d)} \\ \vdots & \ddots & \vdots \\ x_d^{(1)} & \dots & x_d^{(n)} \end{bmatrix} \begin{bmatrix} t^{(1)} \\ \vdots \\ t^{(n)} \end{bmatrix} \Bigg) \\ + \lambda \begin{bmatrix} w^{(1)} & \dots & w^{(d)} \end{bmatrix} \begin{bmatrix} w^{(1)} \\ \vdots \\ w^{(d)} \end{bmatrix}\end{aligned}$$

and now it's gradient:

$$\begin{aligned}\nabla\hat{R}(\theta) &= [\mathbf{X}^T\mathbf{X}\mathbf{w} - \mathbf{X}^T\mathbf{t}] + \lambda(2\mathbf{w}) \\ &= \Bigg(\begin{bmatrix} x_1^{(1)} & \dots & x_1^{(d)} \\ \vdots & \ddots & \vdots \\ x_d^{(1)} & \dots & x_d^{(n)} \end{bmatrix} \begin{bmatrix} x_1^{(1)} & \dots & x_d^{(1)} \\ \vdots & \ddots & \vdots \\ x_1^{(d)} & \dots & x_d^{(n)} \end{bmatrix} \begin{bmatrix} w^{(1)} \\ \vdots \\ w^{(d)} \end{bmatrix} - \begin{bmatrix} x_1^{(1)} & \dots & x_1^{(d)} \\ \vdots & \ddots & \vdots \\ x_d^{(1)} & \dots & x_d^{(n)} \end{bmatrix} \begin{bmatrix} t^{(1)} \\ \vdots \\ t^{(n)} \end{bmatrix} \Bigg) \\ &+ \lambda\Bigg(2\begin{bmatrix} w^{(1)} \\ \vdots \\ w^{(d)} \end{bmatrix}\Bigg)\end{aligned}$$

2.4. *Derive a matrix formulation of the analytical solution to the ridge regression minimization problem by expressing that the gradient is null at the optimum. What happens when $N < d$ and $\lambda = 0$?*

The analytical solution to this problem is defined by the normal equation. To find this, we set the previously-found gradient of the ridge regression problem:

$$\nabla(RSS(w) + \lambda||w||^2) = [\mathbf{X}^T\mathbf{X}\mathbf{w} - \mathbf{X}^T\mathbf{t}] + \lambda(2\mathbf{w}) \tag{2.5}$$

And we set this to zero to obtain:

$$\mathbf{X}^T\mathbf{X}\mathbf{w} + \lambda(2\mathbf{w}) = \mathbf{X}^T\mathbf{t} \tag{2.6}$$

Let **G** be the Gram matrix:

$$\mathbf{G} = \begin{bmatrix} x_1^{(1)} & \dots & x_1^{(d)} \\ \vdots & \ddots & \vdots \\ x_d^{(1)} & \dots & x_d^{(n)} \end{bmatrix} \begin{bmatrix} x_1^{(1)} & \dots & x_d^{(1)} \\ \vdots & \ddots & \vdots \\ x_1^{(d)} & \dots & x_d^{(n)} \end{bmatrix}$$

Then we can write:

$$\begin{aligned} \hat{\mathbf{w}}_{\mathbf{OLS}} &= (\lambda \mathbf{I}_D + \mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{t} \\ &= \Big(\lambda \begin{bmatrix} 1_1^{(1)} & 0 & \dots & 0 & 0 \\ 0 & 1_2^{(1)} & \ddots & \dots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & \ddots & 1_{d-1}^{(d-1)} & 0 \\ 0 & 0 & \dots & 0 & 1_d^{(d)} \end{bmatrix} + \mathbf{G}\Big)^{-1} \begin{bmatrix} x_1^{(1)} & \dots & x_1^{(d)} \\ \vdots & \ddots & \vdots \\ x_d^{(1)} & \dots & x_d^{(n)} \end{bmatrix} \begin{bmatrix} t^{(1)} \\ \vdots \\ t^{(n)} \end{bmatrix} \end{aligned}$$

Solving this equation gives the optimal settings for the weight matrix $\hat{\mathbf{w}}$.

If $n < d$ the Gram matrix $\mathbf{X}^T\mathbf{X}$, which is $d \times d$, has a rank lower than n:

$$rk(\mathbf{X}^T\mathbf{X} \leq n << d) \tag{2.7}$$

Then the Gram matrix is not invertable (another step in our equation) because most of the eigenvalues will likely to be zero! Therefore, the equation cannot be solved analytically, making variable estimation and selection complex.

If we add a value to the diagonal of the Gram matrix, as in Ridge regression ($\lambda\mathbf{I}_D + \mathbf{X}^T\mathbf{X}$) with $\lambda > 0$, this can inflate the independence between the variables in the Gram matrix, making it 'better conditioned' and more likely to be invertable than $\mathbf{X}^T\mathbf{X}$. Hence, when $\lambda = 0$, the inversion necessary for variable estimation becomes complex, even impossible in many cases.

3. Nonlinear Pre-Procesing

3.1. *Give a detailed explanation of $f(x)$ when we us the nonlinear transform $\phi(x)$ on our inputs first.*

$$\begin{aligned} \widetilde{f}_k(\mathbf{x}) &= \sum_{i=1}^{k} \mathbf{w}_i\mathbf{x}^i + b && (3.1) \\ &= \mathbf{w}^T\phi_k(\mathbf{x}) + b && (3.2) \end{aligned}$$

3.2. *Give a detailed explanation of the parameters and their dimensions.*

Let k be the degree of the polynomial transformation generated by the function $\phi_k(x)$. The number of parameters in a normal linear regression is $\theta \in \mathbb{R}^{d+1}$, where we add 1 for the bias term. In this case, we start with an x of $d = 1$, but $\phi(x)$ expands it to be k dimensions, so we end up with a final dimensionality of $\theta \in \mathbb{R}^{k+1}$. In general, the dimensionality is $\theta \in \mathbb{R}^{dk+1}$

3.3. *In dimension $d >= 2$, a polynomial transformation should include not only the individual variable exponents x_i^j , for powers $j <= k$, and variables $i <= d$, but also all the interaction terms of order k and less between several variables (e.g. terms like $x_i^{j_1} x_l^{j_2}$, for $j_1 + j_2 <= k$ and variables i, $l <= d$). For $d = 2$, write down as a function of each of the 2 components of x the transformations $\phi_{poly^1}(x)$, $\phi_{poly^2}(x)$, and $\phi_{poly^3}(x)$.*

In this case, all subscripts all denote the dimension of $\mathbf{x}$, and superscripts denote powers. In the case where $d = 2$:

$$\phi_{poly^1}(x) = \{x_1, x_2\} \tag{3.3}$$

$$\phi_{poly^2}(x) = \{x_1, x_2, x_1x_2, x_1^2, x_2^2\} \tag{3.4}$$

$$\phi_{poly^3}(x) = \{x_1, x_2, x_1^2, x_2^2, x_1x_2, x_1x_2^2, x_1^2x_2, x_1^3, x_2^3\} \tag{3.5}$$

So $\phi_{poly^1}(x)$ gives 2 terms, $\phi_{poly^2}(x)$ gives 6 terms, and $\phi_{poly^3}(x)$ gives 9 terms.

3.4. *What is the dimensionality of $\phi_{poly^k}(x)$, as a function of d and k?*

Let p be the degree of polynomial: $1, 2, ..., k$.

$$\sum_{p=1}^{k} \binom{(d-1)+p}{p} \tag{3.6}$$

Université de Montréal
Email address: joseph@viviano.ca, lea.ricard@umontreal.ca

Practical Part: Linear Regression

October 16, 2018

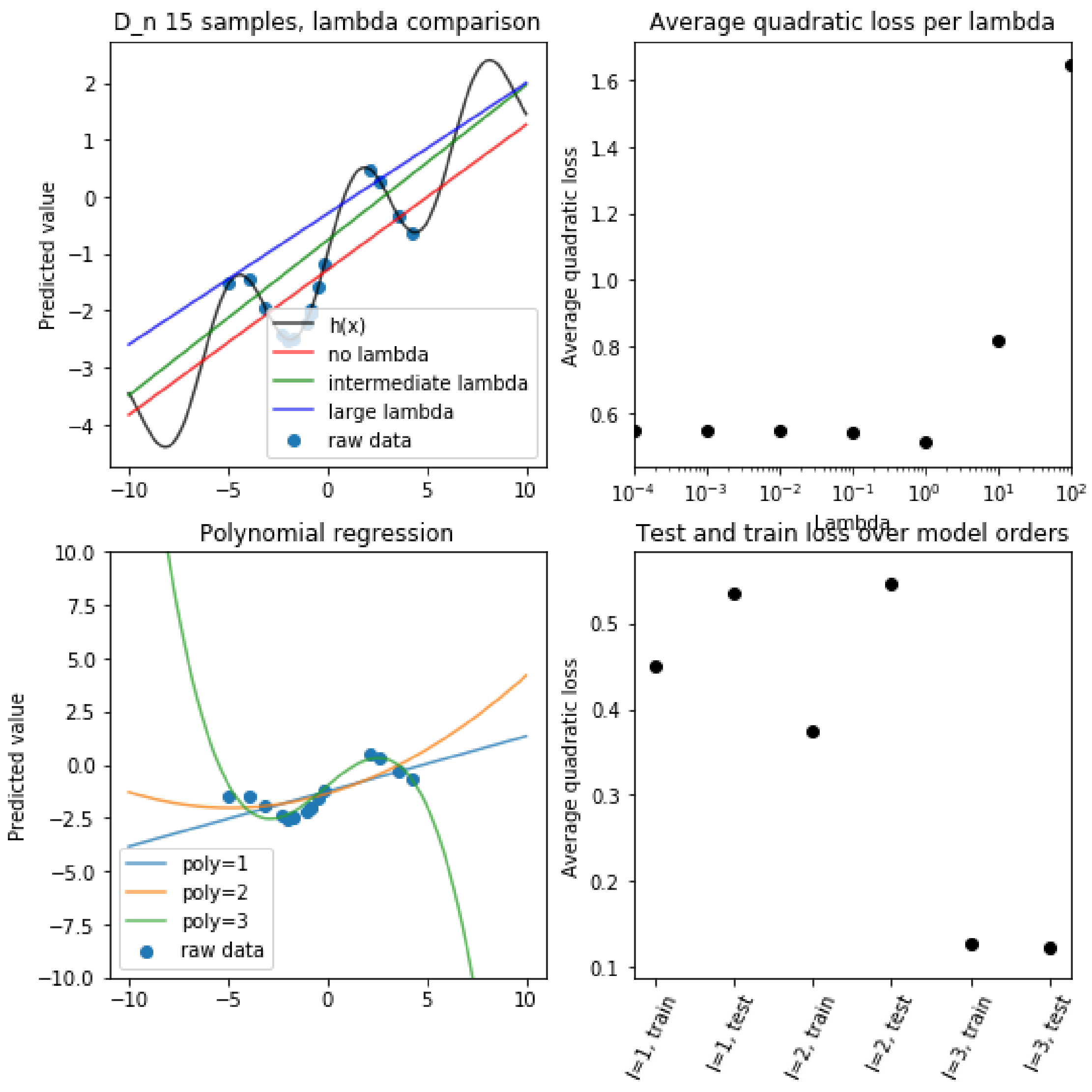
```
In [35]: #!/usr/bin/env python
         import sys, os
         import numpy as np
         from random import uniform
         import math
         get_ipython().run_line_magic('pylab', 'inline')
         import matplotlib.pyplot as plt

         class regression_gradient:
             def __init__(self, lamb=100, step_size=1, n_steps=100):
                 self.lamb = lamb
                 self.step_size = step_size
                 self.n_steps = n_steps

             def train(self, X, y):
                 """Question 1: ridge regression with gradient descent"""

                 # if X is a vector, add dummy dimension
                 if np.ndim(X) == 1:
                     X = np.expand_dims(X, axis=1)

                 stopping_tolerance = 1e-15

                 self.n = np.shape(X)[0]
                 self.d = np.shape(X)[1]
                 # includes b at position 0
                 self.w = np.random.uniform(low=-0.01, high=0.01, size=self.d + 1)

                 # include column of 1s for b
                 X = np.hstack((np.ones((self.n, 1)), X))

                 empirical_risk = np.zeros((self.d + 1))
                 for i in range(self.n_steps):
                     empirical_risk = X.T.dot(X).dot(self.w) - X.T.dot(y)
                     regularization = self.lamb * 2 * self.w
                     gradient = (empirical_risk + regularization)
```

```
            # stopping criteria
            if np.sum(np.abs(gradient)) < stopping_tolerance:
                break

            self.w -= self.step_size  * gradient


    def predict(self, X):
        n = X.shape[0]
        y = np.zeros(n)
        w = self.w[1:]
        b = self.w[0]

        for j in range(n):
            y[j] = w.T.dot(X[j]) + b

        return(y)


def sample_h(n):
    """
    Question 2.
    h(x) = sin(x) + 0.3x -1
    Returns dataset D (x, h(x)) with n points. x in [-5, 5].
    """
    D = np.zeros((n,2))
    for i in range(n):
        D[i,0] = uniform(-5,5)
        D[i,1] = math.sin(D[i,0]) + 0.3*(D[i,0]) - 1
    return D


# plottng options
n_bins = 100
axes_min = -10
axes_max = 10
alpha = 0.75
x = np.atleast_2d(np.linspace(axes_min, axes_max, n_bins)).T

fig, axs = plt.subplots(2, 2, figsize=(9, 9))

# question 3
# plot data
D = sample_h(15)
axs[0][0].scatter(D[:,0], D[:,1]) # raw data

# plot h(x)
y1 = np.zeros(n_bins)
```

```
for i in range(n_bins):
    y1[i] = math.sin(x[i]) + 0.3*(x[i]) - 1
axs[0][0].plot(x, y1, color='black', alpha=alpha)

# training settings
step_size = 5e-5
n_steps = 10000

# plot regression_gradient --lambda 0
mdl_1 = regression_gradient(lamb=0, step_size=step_size, n_steps=n_steps)
mdl_1.train(D[:,0], D[:,1])
axs[0][0].plot(x, mdl_1.predict(x), color='red', alpha=alpha)

# question 4
# plot regression_gradient - lambda intermediate
mdl_2 = regression_gradient(lamb=5, step_size=step_size, n_steps=n_steps)
mdl_2.train(D[:,0], D[:,1])
axs[0][0].plot(x, mdl_2.predict(x), color='green', alpha=alpha)

# plot regression_gradient - lambda large
mdl_3 = regression_gradient(lamb=25, step_size=step_size, n_steps=n_steps)
mdl_3.train(D[:,0], D[:,1])
axs[0][0].plot(x, mdl_3.predict(x), color='blue', alpha=alpha)

axs[0][0].legend(['h(x)',
                  'no lambda',
                  'intermediate lambda',
                  'large lambda',
                  'raw data'])

axs[0][0].set_ylabel('Predicted value')
axs[0][0].set_title('D_n 15 samples, lambda comparison')

# question 5
D_test = sample_h(100)
lambs = [0.0001, 0.001, 0.01, 0.1, 1, 10, 100]

avg_loss = np.zeros(len(lambs))
for i, lamb in enumerate(lambs):
    mdl = regression_gradient(lamb=lamb , step_size=step_size, n_steps=n_steps)
    mdl.train(D[:,0], D[:,1])

    b= mdl.w[0]
    w= mdl.w[1:]

    loss = 0
    for j in range(len(D_test)):
        loss += (np.take(((w.T.dot(D_test[j,0]) + b) - D_test[j,1]), 0))**2
    avg_loss[i] = loss/len(D_test)

# bar plot of average quadratic loss per lambda
width = 1.0
axs[0][1].scatter(lambs, avg_loss, c='black' )
axs[0][1].set_ylabel('Average quadratic loss')
axs[0][1].set_xlabel('Lambda')
axs[0][1].set_xscale('log')
axs[0][1].set_xlim([lambs[0], lambs[-1]])
axs[0][1].set_title('Average quadratic loss per lambda')

# question 6
# training settings
step_size = 5e-6
n_steps = 50000

orders = [1, 2, 3]

loss_test = np.zeros((100, len(orders)))
loss_train = np.zeros((15, len(orders)))

for i, l in enumerate(orders):

    # polynomial preprocessing
    poly = np.zeros((np.shape(D)[0], i+1))
    for j in range(np.shape(D)[0]):
        poly[j] = np.array([D[j, 0]**exp for exp in orders[0:i+1]])

    poly_test = np.zeros((np.shape(D_test)[0], i+1))
    for j in range(np.shape(D_test)[0]):
        poly_test[j] = np.array([D_test[j, 0]**exp for exp in orders[0:i+1]])

    # ridge regression
    mdl = regression_gradient(lamb=0.01, step_size=step_size, n_steps=n_steps)
    mdl.train(poly, D[:,1])
    b = mdl.w[0]
    w = mdl.w[1:]

    # ploting f learned with ridge regression
    y = np.zeros(n_bins)
    for z in range(n_bins):
        processed_x = np.array([x[z]**exp for exp in orders[:i+1]])
        y[z] = np.dot(w.T, processed_x) + b
    axs[1][0].plot(x, y, alpha=alpha)

    # collect empirical risk, true risk
    train_predictions = mdl.predict(poly)
    test_predictions = mdl.predict(poly_test)
```

```
        for j in range(len(D_test)):
            loss_test[j, i] = (np.take(((w.T.dot(poly_test[j, :]) + b) - D_test[j,1]), 0)

        for j in range(len(D)):
            loss_train[j, i] = (np.take(((w.T.dot(poly[j, :]) + b) - D[j,1]), 0))**2

    # print models
    axs[1][0].scatter(D[:,0], D[:,1])
    axs[1][0].legend(['poly=1', 'poly=2', 'poly=3', 'raw data'])
    axs[1][0].set_ylabel('Predicted value')
    axs[1][0].set_ylim([-10, 10])
    axs[1][0].set_title('Polynomial regression')

    # print losses
    axs[1][1].scatter(np.array([1,2,3,4,5,6]), np.array([
        np.mean(loss_train[:, 0]), np.mean(loss_test[:, 0]),
        np.mean(loss_train[:, 1]), np.mean(loss_test[:, 1]),
        np.mean(loss_train[:, 2]), np.mean(loss_test[:, 2])
    ]), c='black')


    axs[1][1].set_xticklabels(['',
                               'l=1, train', 'l=1, test',
                               'l=2, train', 'l=2, test',
                               'l=3, train', 'l=3, test'])
    for tick in axs[1][1].get_xticklabels():
        tick.set_rotation(65)
    axs[1][1].set_title('Test and train loss over model orders')
    axs[1][1].set_ylabel('Average quadratic loss')

    plt.show()

    plt.savefig('report.jpg')
```

```
Populating the interactive namespace from numpy and matplotlib


/opt/python/sci_36/lib/python3.6/site-packages/IPython/core/magics/pylab.py:160: UserWarning: p
`%matplotlib` prevents importing * from pylab and numpy
  "\n`%matplotlib` prevents importing * from pylab and numpy"
```



```
<matplotlib.figure.Figure at 0x7f3c6edc41d0>
```

As l increases, the empirical risk stays steady for l=1 and l=2, and drops dramatically for l=3, since this particular polynomial fits the function generating the data fairly well. Increasing model order does not help the test performance for l=1 and l=2. Since l=3 is a good model for this particular dataset, the test performance is also good for this model. We predict that as one adds more orders to the polynomial expansion, empirical risk will stay low, however, the true risk (performance on test set) will become worse. This is because our model will become overfit to the training data for l > 3.