Homework3_DS-GA1013

February 27, 2021

Name: Zhuoyuan Xu (Kallen Xu)

NetID: zx1137

Due Date: Feb 28, 2021

Problem 1

From our inclass materials, the ridge regression estimate can be written as

$$\begin{aligned}\beta_{RR} &= arg\min_{\beta} ||y - X^T\beta||_2^2 + \lambda||\beta||_2^2 \\ &= arg\min_{\beta} || \begin{bmatrix} y \\ 0 \end{bmatrix} - \begin{bmatrix} X^T \\ \sqrt{\lambda} I \end{bmatrix} \beta||_2^2 \\ &= (XX^T + \lambda I)^{-1} Xy\end{aligned}$$

and the ordinary least square estimate cna be written as

$$\begin{aligned}\beta_{OLS} &= arg\min_{\beta} ||y - X^T\beta||_2^2 \\ &= (XX^T)^{-1} Xy\end{aligned}$$

It can be observed that ridge regression is equivalent to solving the formula

$$\beta_{RR} = (\tilde{X}^T\tilde{X})^{-1}\tilde{X}\tilde{y}$$

with augmented data sets $\tilde{X} = \begin{bmatrix} X \\ \sqrt{\lambda} I_{p\times p} \end{bmatrix}$ and $\tilde{y} = \begin{bmatrix} y \\ 0_{p\times 1} \end{bmatrix}$

The addditional examples in the augmented dataset add p rows to X and y. X has value $\sqrt{\lambda}$ on the diagonal of the additional rows; other entries in these rows in X and y have value 0.

Meanwhile,

$$\tilde{X}^T\tilde{X} = X^TX + \lambda I$$

and

$$\tilde{X}^T\tilde{y} = X^Ty$$

By changing the value of λ in the additional rows, the β_{RR} can converge to β_{OLS} (if $\lambda \to 0$) or 0 (if $\lambda \to \infty$). They controls the trade-off between bias and variance to achieve the best test error.

Problem 2

(a) The OLS estimator can be written as

$$\begin{aligned}\beta_{OLS} &= (XX^T)^{-1}Xy \\ &= (\begin{bmatrix} w_1^T \\ w_2^T \end{bmatrix} \begin{bmatrix} w_1 & w_2 \end{bmatrix})^{-1} \begin{bmatrix} w_1^T \\ w_2^T \end{bmatrix} y \\ &= (\begin{bmatrix} w_1^T w_1 & w_1^T w_2 \\ w_2^T w_1 & w_2^T w_2 \end{bmatrix})^{-1} \begin{bmatrix} w_1^T \\ w_2^T \end{bmatrix} y\end{aligned}$$

In this problem, y is defined as

$$y = \beta_{true} w_1 + z$$

Meanwhile, the vectors w_1, w_2, $w_\perp$ and z all have unit l_2 norm. $w_\perp$ and w_1 are perpendicular to each other.

Also, we know that $w_2 = \alpha w_1 + \sqrt{1-\alpha^2} w_\perp$, with $w_1^T z = 0.1$ and $w_\perp^T z = 0.1$

Thus,

$$\begin{aligned}\beta_{OLS} &= (\begin{bmatrix} w_1^T w_1 & w_1^T w_2 \\ w_2^T w_1 & w_2^T w_2 \end{bmatrix})^{-1} \begin{bmatrix} w_1^T \\ w_2^T \end{bmatrix} (\beta_{true} w_1 + z) \\ &= (\begin{bmatrix} w_1^T w_1 & w_1^T w_2 \\ w_2^T w_1 & w_2^T w_2 \end{bmatrix})^{-1} \begin{bmatrix} \beta_{true} w_1^T w_1 + w_1^T z \\ \beta_{true} w_2^T w_1 + w_2^T z \end{bmatrix} \\ &= (\begin{bmatrix} 1 & \alpha \\ \alpha & 1 \end{bmatrix})^{-1} \begin{bmatrix} \beta_{true} + w_1^T z \\ \alpha\beta_{true} + w_2^T z \end{bmatrix} \\ &= (\begin{bmatrix} 1 & \alpha \\ \alpha & 1 \end{bmatrix})^{-1} \begin{bmatrix} \beta_{true} + w_1^T z \\ \alpha\beta_{true} + (\alpha w_1 + \sqrt{1-\alpha^2} w_\perp)^T z \end{bmatrix} \\ &= (\begin{bmatrix} 1 & \alpha \\ \alpha & 1 \end{bmatrix})^{-1} \begin{bmatrix} \beta_{true} + 0.1 \\ \alpha\beta_{true} + 0.1(\alpha + \sqrt{1-\alpha^2}) \end{bmatrix}\end{aligned}$$

In this equation,

$$(\begin{bmatrix} 1 & \alpha \\ \alpha & 1 \end{bmatrix})^{-1} = \frac{1}{1-\alpha^2} \begin{bmatrix} 1 & -\alpha \\ -\alpha & 1 \end{bmatrix}$$

Then,

$$\begin{aligned}\beta_{OLS} &= \frac{1}{1-\alpha^2} \begin{bmatrix} 1 & -\alpha \\ -\alpha & 1 \end{bmatrix} \begin{bmatrix} \beta_{true} + 0.1 \\ \alpha\beta_{true} + 0.1(\alpha + \sqrt{1-\alpha^2}) \end{bmatrix} \\ &= \frac{1}{1-\alpha^2} \begin{bmatrix} \beta_{true} + 0.1 - \alpha^2\beta_{true} - 0.1\alpha(\alpha + \sqrt{1-\alpha^2}) \\ -\alpha\beta_{true} - 0.1\alpha + \alpha\beta_{true} + 0.1(\alpha + \sqrt{1-\alpha^2}) \end{bmatrix} \\ &= \frac{1}{1-\alpha^2} \begin{bmatrix} (1-\alpha^2)\beta_{true} + 0.1 - 0.1\alpha(\alpha + \sqrt{1-\alpha^2}) \\ 0.1\sqrt{1-\alpha^2} \end{bmatrix} \\ &= \begin{bmatrix} \beta_{true} + 0.1 - \frac{0.1\alpha\sqrt{1-\alpha^2}}{1-\alpha^2} \\ \frac{0.1\sqrt{1-\alpha^2}}{1-\alpha^2} \end{bmatrix}\end{aligned}$$

When $\alpha \to 1$, $w_2 \to w_1$. The term $\frac{1}{1-\alpha^2}$ will have the denominator approaching 0. Also, the matrix multiplication within the parenthesis may not be invertible, or becomes very huge with eigenvalues close to 0. The ordinary least squares estimator therefore does not exist or is unstable.

(b) The corresponding estimate of the response

$$\begin{aligned} y_{OLS} &= X^T\beta_{OLS} \\ &= \begin{bmatrix} w_1 & \alpha w_1 + \sqrt{1-\alpha^2}w_\perp \end{bmatrix} \begin{bmatrix} \beta_{true} + 0.1 - \frac{0.1\alpha\sqrt{1-\alpha^2}}{1-\alpha^2} \\ \frac{0.1\sqrt{1-\alpha^2}}{1-\alpha^2} \end{bmatrix} \\ &= \begin{bmatrix} \beta_{true}w_1 + 0.1w_1 + 0.1w_\perp \end{bmatrix} \end{aligned}$$

When $\alpha \to 1$, $w_2 \to w_1$,

$$y_{OLS} = \begin{bmatrix} \beta_{true}w_1 + 0.1w_1 + 0.1w_\perp \end{bmatrix}$$

since the term $w_\perp$ is in the final expression, and it is not collinear with the true feature.

(c) The ridge regression estimator is

$$\begin{aligned} \beta_{RR} &= (XX^T + \lambda I)^{-1}Xy \\ &= (\begin{bmatrix} w_1^T \\ w_2^T \end{bmatrix} \begin{bmatrix} w_1 & w_2 \end{bmatrix} + \begin{bmatrix} \lambda & 0 \\ 0 & \lambda \end{bmatrix})^{-1} \begin{bmatrix} w_1^T \\ w_2^T \end{bmatrix} y \\ &= (\begin{bmatrix} 1+\lambda & \alpha \\ \alpha & 1+\lambda \end{bmatrix})^{-1} \begin{bmatrix} w_1^T \\ (\alpha w_1 + \sqrt{1-\alpha^2}w_\perp)^T \end{bmatrix} (\beta_{true}w_1 + z) \\ &= (\begin{bmatrix} 1+\lambda & \alpha \\ \alpha & 1+\lambda \end{bmatrix})^{-1} \begin{bmatrix} \beta_{true} + 0.1 \\ \alpha\beta_{true} + 0.1(\alpha + \sqrt{1-\alpha^2}) \end{bmatrix} \end{aligned}$$

In this equation,

$$(\begin{bmatrix} 1+\lambda & \alpha \\ \alpha & 1+\lambda \end{bmatrix})^{-1} = \frac{1}{(1+\lambda)(1+\lambda) - \alpha^2} \begin{bmatrix} 1+\lambda & -\alpha \\ -\alpha & 1+\lambda \end{bmatrix}$$

Thus,

$$\begin{aligned} \beta_{RR} &= \frac{1}{(1+\lambda)(1+\lambda) - \alpha^2} \begin{bmatrix} 1+\lambda & -\alpha \\ -\alpha & 1+\lambda \end{bmatrix} \begin{bmatrix} \beta_{true} + 0.1 \\ \alpha\beta_{true} + 0.1(\alpha + \sqrt{1-\alpha^2}) \end{bmatrix} \\ &= \frac{1}{(1+\lambda)^2 - \alpha^2} \begin{bmatrix} (1+\lambda-\alpha^2)\beta_{true} + 0.1(1+\lambda-\alpha^2-\alpha\sqrt{1-\alpha^2}) \\ \lambda\alpha\beta_{true} + 0.1(1+\lambda)(\alpha+\sqrt{1-\alpha^2}) - 0.1\alpha \end{bmatrix} \end{aligned}$$

When $\alpha \to 1$, $w_2 \to w_1$. If we plug this into the equation then

$$\beta_{RR} = \frac{1}{\lambda^2 + 2\lambda} \begin{bmatrix} \lambda\beta_{true} + 0.1\lambda \\ \lambda\beta_{true} + 0.1\lambda \end{bmatrix}$$

which, different from OLS, will not lead to 0 denominator for the chosen penalty $\lambda > 0$ (positive definite) and thus the matrix multiplication is always invertible.

(d) The corresponding estimate of the response would be

$$\begin{aligned} y_{RR} &= X^T\beta_{RR} \\ &= \frac{1}{(1+\lambda)(1+\lambda) - \alpha^2} \begin{bmatrix} w_1 & \alpha w_1 + \sqrt{1-\alpha^2}w_\perp \end{bmatrix} \begin{bmatrix} (1+\lambda-\alpha^2)\beta_{true} + 0.1(1+\lambda-\alpha^2-\alpha\sqrt{1-\alpha^2}) \\ \lambda\alpha\beta_{true} + 0.1(1+\lambda)(\alpha+\sqrt{1-\alpha^2}) - 0.1\alpha \end{bmatrix} \end{aligned}$$

When $\alpha \to 1$, $w_2 \to w_1$

$$\begin{aligned} y_{RR} &= \frac{1}{\lambda^2 + 2\lambda} \begin{bmatrix} w_1 & w_1 \end{bmatrix} \begin{bmatrix} \lambda\beta_{true} + 0.1\lambda \\ \lambda\beta_{true} + 0.1\lambda \end{bmatrix} \\ &= \frac{2w_1}{\lambda + 2} \left[\beta_{true} + 0.1\right] \end{aligned}$$

and it is collinear with the true feature w_1.

Problem 3

(a) In ridge regression, we shrink the coefficients that has minor contribution to the prediction by adding an extra penalty term. If in this problem, we have a prior knowledge that the coefficients should be close to a certain value β_{prior}, we can penalize the coefficients that are too far from this knowledge. The optimization problem then becomes

$$\beta_{RR} = arg \min_{\beta} ||y - X^T\beta||_2^2 + \lambda||\beta - \beta_{prior}||_2^2$$

(b) The original ridge regression cost function can be written in the closed form as

$$(y - X^T\beta)^T(y - X^T\beta) + \lambda(\beta - \beta_{prior})^T(\beta - \beta_{prior})$$

If we take the first derivative and set the value to 0,

$$2(XX^T)\beta - 2Xy + 2\lambda(\beta - \beta_{prior}) = 0$$

Thus we can get

$$\begin{aligned}\beta_{RR} &= (XX^T - \lambda I)^{-1}(Xy + \lambda\beta_{prior})\\ &= (XX^T - \lambda I)^{-1}(XX^T\beta_{true} + X\tilde{z} + \lambda\beta_{prior})\end{aligned}$$

If we perform SVD onto the expression we can get

$$\begin{aligned}\beta_{RR} &= (XX^T - \lambda I)^{-1}(XX^T\beta_{true} + X\tilde{z} + \lambda\beta_{prior})\\ &= (US^2U^T + \lambda UU^T)^{-1}(US^2U^T\beta_{true} + USV^T\tilde{z} + \lambda\beta_{prior})\\ &= U(S^2 + \lambda I)^{-1}U^T(US^2U^T\beta_{true} + USV^T\tilde{z} + \lambda\beta_{prior})\\ &= U(S^2 + \lambda I)^{-1}S^2U^T\beta_{true} + U(S^2 + \lambda I)^{-1}SV^T\tilde{z} + U(S^2 + \lambda I)^{-1}U^T\lambda\beta_{prior}\end{aligned}$$

Since assume the mean of $\tilde{z}$ is 0, the mean of this expression is

$$\begin{aligned}E(\beta_{RR}) &= E(U(S^2 + \lambda I)^{-1}S^2U^T\beta_{true} + U(S^2 + \lambda I)^{-1}SV^T\tilde{z} + U(S^2 + \lambda I)^{-1}U^T\lambda\beta_{prior})\\ &= E(U(S^2 + \lambda I)^{-1}S^2U^T\beta_{true}) + E(U(S^2 + \lambda I)^{-1}U^T\lambda\beta_{prior}))\\ &= \beta_{bias} + \sum_{j=1}^{p} \frac{\lambda < u_j, \beta_{prior} >}{s_j^2 + \lambda} u_j\end{aligned}$$

However, this term will not affect the variance of the ridge regression since we are only adding an extra term which is not a function of β to the expectation. This can also be shown by observing the equation for variance in Theorem 4.3 in the note.

(c) If we have the prior knowledge that the coefficients are close to β_{prior}, instead of initialize at origin, we can initialize at β_{prior}. Given the linear model, the gradient descent updates can be written as

$$\beta^{(k+1)} = (I - \alpha XX^T)\beta^{(k)} + \alpha Xy$$

The initial step is

$$\beta^{(0)} = \beta_{prior}$$

Then,

$$\begin{aligned}\beta^{(1)} &= (I - \alpha XX^T)\beta^{(0)} + \alpha Xy \\ &= (I - \alpha XX^T)\beta_{prior} + \alpha Xy\end{aligned}$$

$$\begin{aligned}\beta^{(2)} &= (I - \alpha XX^T)\beta^{(1)} + \alpha Xy \\ &= (I - \alpha XX^T)((I - \alpha XX^T)\beta_{prior} + \alpha Xy) + \alpha Xy \\ &= (I - \alpha XX^T)^2\beta_{prior} + (I - \alpha XX^T)\alpha Xy + \alpha Xy\end{aligned}$$

$$\begin{aligned}\beta^{(3)} &= (I - \alpha XX^T)\beta^{(2)} + \alpha Xy \\ &= (I - \alpha XX^T)((I - \alpha XX^T)^2\beta_{prior} + (I - \alpha XX^T)\alpha Xy + \alpha Xy) + \alpha Xy \\ &= (I - \alpha XX^T)^3\beta_{prior} + (I - \alpha XX^T)^2\alpha Xy + (I - \alpha XX^T)\alpha Xy + \alpha Xy\end{aligned}$$

To generalize, the expression of the update equation will be

$$\beta^{(k+1)} = (I - \alpha XX^T)^{k+1}\beta_{prior} + \sum_{i=0}^{k}(I - \alpha XX^T)^i\alpha Xy$$

(d) For the following calculation, assume X has full rank and $n \geq p$, if we apply singular value decomposition to the expression we can get

$$\begin{aligned}\beta^{(k+1)} &= (UU^T - \alpha US^2U^T)^{k+1}\beta_{prior} + \alpha\sum_{i=0}^{k}(UU^T - \alpha US^2U^T)^i USV^Ty \\ &= Udiag_{j=1}^{p}((1 - \alpha s_j^2)^{k+1})U^T\beta_{prior} + Udiag_{j=1}^{p}(\frac{1 - (1 - \alpha s_j^2)^{k+1}}{s_j^2})V^Ty\end{aligned}$$

Then plug in the value for y

$$\begin{aligned}\beta^{(k+1)} &= Udiag_{j=1}^{p}((1 - \alpha s_j^2)^{k+1})U^T\beta_{prior} + Udiag_{j=1}^{p}(\frac{1 - (1 - \alpha s_j^2)^{k+1}}{s_j^2})V^T(X^T\beta_{true} + \tilde{z}) \\ &= Udiag_{j=1}^{p}((1 - \alpha s_j^2)^{k+1})U^T\beta_{prior} + Udiag_{j=1}^{p}(1 - (1 - \alpha s_j^2)^{k+1})U^T\beta_{true} \\ &+ Udiag_{j=1}^{p}(\frac{1 - (1 - \alpha s_j^2)^{k+1}}{s_j^2})V^T\tilde{z}\end{aligned}$$

Given the mean of $\tilde{z}$ to be 0, the mean is thus changed

$$E(\beta^{(k+1)}) = Udiag_{j=1}^{p}((1 - \alpha s_j^2)^{k+1})U^T\beta_{prior} + Udiag_{j=1}^{p}(1 - (1 - \alpha s_j^2)^{k+1})U^T\beta_{true}$$

Similarly, the variance will not be affected since we are only adding an extra term which does not affect the noise.

Problem 4

(a) The objective function

$$J(\beta) = \frac{1}{m}\sum_{i=1}^{m}(\beta^T x_i - y_i)^2$$

and we can set $f_i(\beta) = (\beta^T x_i - y_i)^2$, so our objective function can be rewritten as

$$J(\beta) = \frac{1}{m}\sum_{i=1}^{m} f_i(\beta)$$

Meanwhile, the stochastic gradient descent objective function is just the $\nabla f_i(\beta)$ for some data point i chosen uniformly at random from {1, ..., m}.

With our objective function written inside the summation, the gradient can be written as

$$\nabla J(\beta) = \nabla \frac{1}{m}\sum_{i=1}^{m} f_i(\beta)$$

Since gradient is a linear operation,

$$\nabla J(\beta) = \frac{1}{m}\sum_{i=1}^{m} \nabla f_i(\beta)$$

Recall the expected value formula, the expected value of this gradient is the sum of all possible choices of i, which is sampled uniformly at random from {1, ..., m}

$$E[\nabla f_i(\beta)] = \nabla \sum_{j=1}^{m} P(j = i) f_i(\beta)$$

We can also move the gradient into the summation,

$$E[\nabla f_i(\beta)] = \frac{1}{m}\sum_{j=1}^{m} \nabla f_i(\beta)$$

Therefore,

$$E[\nabla f_i(\beta)] = \nabla J(\beta)$$

the SGD gradient is an unbiased estimator of the real gradient.

(b) From the previous problems, we can write the gradient of our $f_i(\beta)$ as

$$\begin{aligned}\nabla f_i(\beta) &= \nabla(\beta^T x_i - y_i)^2 \\ &= 2(\beta^T x_i - y_i)\nabla(\beta^T x_i - y_i) \\ &= 2(\beta^T x_i - y_i)x_i\end{aligned}$$

The expression for updating β in the gradient descent algorithm for a step size η is

$$\beta^{i+1} = \beta^i - \eta^i \nabla J(\beta) = \beta^i - \eta^i \nabla f_i(\beta)$$

then we can plug in our formula for $\nabla f_i(\beta)$

$$\beta^{i+1} = \beta^i - \eta^i \nabla J(\beta) = \beta^i - \eta^i(2((\beta^i)^T x_i - y_i)x_i)$$

(c)

[1]:
```
str_path = "./weather/"
```

[2]:
```
import matplotlib.pyplot as plt
import numpy as np
from os import listdir
from sklearn import linear_model

np.random.seed(2021)
```

[3]:
```
def extract_temp(file_name,col_ind):
    data_aux = np.loadtxt(file_name, usecols=range(10))
    data = data_aux[:,col_ind]
    err_count = 0
    ind_errs = []
    for ind in range(data.shape[0]):
        if data[ind] > 100 or data[ind] < -100:
            err_count = err_count + 1
            ind_errs.append(ind)
            data[ind] = data[ind-1]
    print("File name: " + file_name)
    print("Errors: " + str(err_count) + " Indices: " + str(ind_errs))
    return data

def create_data_matrix(str_path):
    file_name_list = listdir(str_path)
    file_name_list.sort()
    col_ind = 8 # 8 = last 5 minutes, 9 = average over the whole hour
    data_matrix = []
    ind = 0
    for file_name in file_name_list:
        if file_name[0] == '.':
            continue
        else:
            print("Station " + str(ind))
            ind = ind + 1
            data_aux = extract_temp(str_path + file_name,col_ind)
            if len(data_matrix) == 0:
                data_matrix = data_aux
            else:
                data_matrix = np.vstack((data_matrix,data_aux))
    return data_matrix.T
```

[4]:
```
load_files = False
if load_files:
    str_path_2015 = str_path + "hourly/2015/"
    data_matrix = create_data_matrix(str_path_2015)
else:
    data_matrix = np.load(str_path +"hourly_temperature_2015.npy")

file_name_list = listdir(str_path + "hourly/2015/")
file_name_list.sort()
```

[5]:
```
ind_response = 18
print("Response is " + str(file_name_list[ind_response]))
y_raw = data_matrix[:,ind_response]
ind_X = np.hstack((np.arange(0,ind_response),np.
 ↪arange(ind_response+1,data_matrix.shape[1])))
X_raw = data_matrix[:,ind_X]
n_features = X_raw.shape[1]
```

```
Response is CRNH0203-2015-AL_Valley_Head_1_SSW.txt
```

[6]:
```
n_test = int(1e3)
n_val = int(1e2)
n_train = data_matrix.shape[0] - n_test - n_val
```

[7]:
```
aux_ind = np.random.permutation(range(data_matrix.shape[0]))
ind_test = aux_ind[:n_test]
ind_val = aux_ind[n_test:(n_test+n_val)]
X_test = X_raw[ind_test,:]
y_test = y_raw[ind_test]
X_val = X_raw[ind_val,:]
y_val = y_raw[ind_val]
ind_train = aux_ind[(n_test+n_val):int(n_test+n_val+n_train)]
X_train = X_raw[ind_train,:]
y_train = y_raw[ind_train]
```

For this problem we will work with features that are zero mean and unit variance. We standardize the data below. Make sure to standardize the validation and test data using the statistics you compute from train data

[8]:
```
center_vec = X_train.mean(axis=0)
X_train_centered = X_train - center_vec
col_norms = np.linalg.norm(X_train_centered, axis=0) / np.sqrt(n_train)
X_train_norm = np.true_divide(X_train_centered, col_norms)

y_train_center = y_train.mean()
y_train_centered = y_train - y_train_center
norm_y_train = np.linalg.norm(y_train_centered) / np.sqrt(n_train)
y_train_norm = y_train_centered / norm_y_train
```

[9]:
```
coef, residuals, rank, singular = np.linalg.lstsq(X_train_norm, y_train_norm,␣
 ↪rcond=None)
```

```
[10]: # standardize validation data
      X_val_centered = X_val - center_vec
      X_val_norm = np.true_divide(X_val_centered, col_norms)

      y_val_centered = y_val - y_train_center
      y_val_norm = y_val_centered / norm_y_train
```

```
[11]: # validation error from np.linalg.lstsq
      y_val_reg = norm_y_train * np.dot(X_val_norm, coef) + y_train_center
      error_val = np.linalg.norm(y_val - y_val_reg) / np.sqrt(len(y_val))
      print('Validation error from np.linalg.lstsq is', error_val)
```

```
Validation error from np.linalg.lstsq is 1.2136898851357365
```

(d)

```
[12]: def compute_stochastic_gradient(X, y, beta):
          grad = (X.dot(beta) - y) * X
          return grad
```

```
[13]: def compute_square_loss(X, y, beta):
          """
          Given a set of X, y, theta, compute the average square loss for predicting␣
       ↪y with X*theta.

          Args:
              X - the feature vector, 2D numpy array of size(num_instances,␣
       ↪num_features)
              y - the label vector, 1D numpy array of size(num_instances)
              theta - the parameter vector, 1D array of size(num_features)

          Returns:
              loss - the average square loss, scalar
          """
          m = y.shape[0]
          loss = np.linalg.norm((beta @ X.T) - y)**2/m
          return loss
```

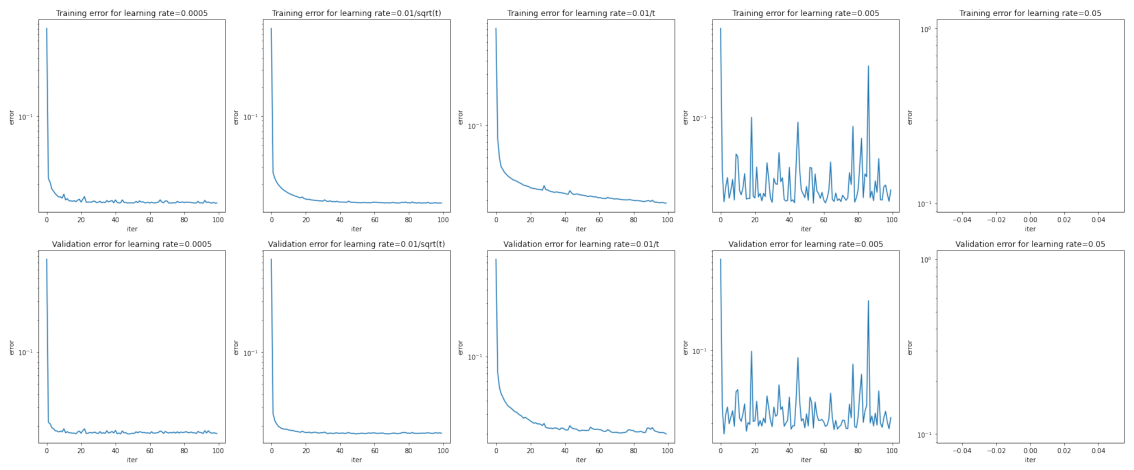
```
[35]: def stochastic_grad_descent(X_train, y_train, X_val, y_val, alpha, num_epoch):
          """
          Args:
              X - the feature vector, 2D numpy array of size (num_instances,␣
       ↪num_features)
              y - the label vector, 1D numpy array of size (num_instances)
              alpha - string or float, step size in gradient descent
                      NOTE: In SGD, it's not a good idea to use a fixed step size.␣
       ↪Usually it's set to 1/sqrt(t) or 1/t
```

```
                if alpha is a float, then the step size in every step is the␣
↪float.
                if alpha == "1/sqrt(t)", alpha = 1/sqrt(t).
                if alpha == "1/t", alpha = 1/t.
        num_epoch - number of epochs to go through the whole training set
    """
    num_instances, num_features = X_train.shape[0], X_train.shape[1]
    beta = 0.001*np.ones(num_features) #Initialize beta

    train_hist = np.zeros((num_epoch, 1)) #Initialize train_hist
    train_hist[0] = compute_square_loss(X_train, y_train, beta)

    val_hist = np.zeros((num_epoch, 1)) #Initialize train_hist
    val_hist[0] = compute_square_loss(X_val, y_val, beta)

    alpha_step = alpha

    for i in range(1, num_epoch):
        randomidx = np.random.permutation(num_instances)

        for t in range(len(randomidx)):
            if alpha == "0.01/sqrt(t)":
                alpha_step = 0.01/np.sqrt(t+1)
            if alpha == "0.01/t":
                alpha_step = 0.01/(t+1)
            if alpha == "0.1/t":
                alpha_step = 0.1/(t+1)
            if alpha == "0.001/t":
                alpha_step = 0.001/(t+1)

            Xi = X_train[randomidx[t], :]
            yi = y_train[randomidx[t]]
            beta = beta - alpha_step * compute_stochastic_gradient(Xi, yi, beta)
        train_hist[i] = compute_square_loss(X_train, y_train, beta)
        val_hist[i] = compute_square_loss(X_val, y_val, beta)

    return train_hist, val_hist, beta
```

[36]:
```
learning_rates = [0.0005, "0.01/sqrt(t)", "0.01/t", 0.005, 0.05]
num_epoch = 100
beta_hist = []
```

[37]:
```
fig, ax = plt.subplots(2, 5, figsize=(30, 12))
for i in range(len(learning_rates)):
    train_hist, val_hist, beta = stochastic_grad_descent(X_train_norm,␣
↪y_train_norm, X_val_norm, y_val_norm, learning_rates[i], num_epoch)
    beta_hist.append(beta)
    ax[0, i].plot(np.arange(0, num_epoch), train_hist)
    ax[0, i].set_yscale('log')
    ax[0, i].set_title('Training error for learning rate={}'.
↪format(learning_rates[i]))
    ax[0, i].set_ylabel('error')
    ax[0, i].set_xlabel('iter')
    ax[1, i].plot(np.arange(0, num_epoch), val_hist)
    ax[1, i].set_yscale('log')
    ax[1, i].set_title('Validation error for learning rate={}'.
↪format(learning_rates[i]))
    ax[1, i].set_ylabel('error')
    ax[1, i].set_xlabel('iter')
```

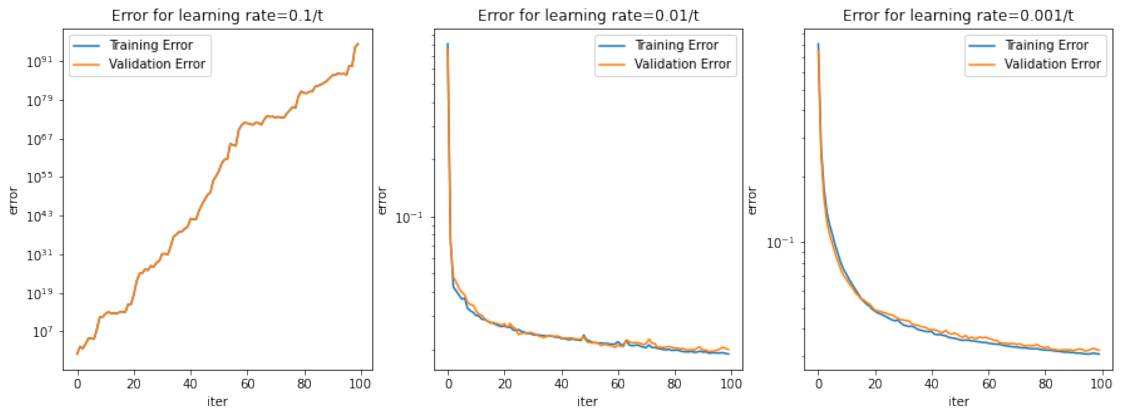
```
<ipython-input-12-bb82a1129a9d>:2: RuntimeWarning: overflow encountered in
multiply
  grad = (X.dot(beta) - y) * X
```



The stochastic gradient descent diverges for step size 0.05, and thus there is an overflow error and no plot it produced.

```
[38]: learning_rates = ["0.1/t", "0.01/t", "0.001/t"]
      fig, ax = plt.subplots(1, 3, figsize=(15, 5))
      for i in range(len(learning_rates)):
          train_hist, val_hist, beta = stochastic_grad_descent(X_train_norm,␣
      ↪y_train_norm, X_val_norm, y_val_norm, learning_rates[i], num_epoch)
          beta_hist.append(beta)
          ax[i].plot(np.arange(0, num_epoch), train_hist, label='Training Error')
          ax[i].legend()
          ax[i].plot(np.arange(0, num_epoch), val_hist, label='Validation Error')
          ax[i].legend()
          ax[i].set_yscale('log')
          ax[i].set_title('Error for learning rate={}'.format(learning_rates[i]))
```

```
    ax[i].set_ylabel('error')
    ax[i].set_xlabel('iter')
plt.show()
```



When we take relatively large and fixed step sizes, it is possible for the SGD algorithm to not converge or even has overflow in calculation. After taking step sizes changing with t, the curves seems to have less noises because the step size shrinks as the algorithms gets closer to the optimal solution. Also, the algorithm with learning rate 0.01/sqrt(t) converges faster than that with 0.01/t because the decreasing rate of the learning rate is lower for the prior one, i.e. the size of the prior learning rate is kept larger than that of the later learning rate as t increases. We can also observe the same trend in step sizes 0.1/t, 0.01t and 0.001/t: 0.1/t diverges because the initial steps are very large, 0.01/t converges faster than 0.001/t but has more noises in the plot.

(e)

[39]:
```
# standardize test data
X_test_centered = X_test - center_vec
X_test_norm = np.true_divide(X_test_centered, col_norms)

y_test_centered = y_test - y_train_center
y_test_norm = y_test_centered / norm_y_train
```

[40]:
```
# test error from np.linalg.lstsq
y_test_reg = norm_y_train * np.dot(X_test_norm, coef) + y_train_center
error_test = np.linalg.norm(y_test - y_test_reg) / np.sqrt(len(y_test))
print('Test error from np.linalg.lstsq is', error_test)
```

```
Test error from np.linalg.lstsq is 1.0809369027614686
```

[43]:
```
# test error from SGD
error_test = []
for i in [0, 1, 2, 3, 5, 6, 7]:
    y_test_reg = norm_y_train * np.dot(X_test_norm, beta_hist[i]) +␣
 ↪y_train_center
```

```
    error_test.append(np.linalg.norm(y_test - y_test_reg) / np.
 ↪sqrt(len(y_test)))
print('Test error from best SGD beta is', np.min(error_test))
```

```
Test error from best SGD beta is 1.0777564522341092
```