Final project type 3

More algorithms

4 조

김두은 김태완 이혜림

순서

Regularization
(Ridge, Lasso)

Stochastic gradient descent

Coordinate descent

Regularization

Y = X β + ε 의 regression model을 fitting : OLS

$$\underset{\beta}{\text{minimize}} \quad \frac{1}{2}\|\mathbf{y} - \mathbf{X}\beta\|_2^2$$

그러나 이 경우, overfitting의 문제가 생길 수 있다 → 'Regularization'

$$\begin{aligned} &\underset{\beta}{\text{minimize}} && \frac{1}{2}\|\mathbf{y} - \mathbf{X}\beta\|_2^2 \\ &\text{subject to} && \|\beta\|_2 \le t \end{aligned}$$

Or

$$\underset{\beta}{\text{minimize}} \quad \frac{1}{2}\|\mathbf{y} - \mathbf{X}\beta\|_2^2 + \lambda\|\beta\|_2^2$$

1) Ridge : β 의 l2-norm으로 제약

$$\begin{aligned} &\underset{\beta}{\text{minimize}} && \frac{1}{2}\|\mathbf{y} - \mathbf{X}\beta\|_2^2 \\ &\text{subject to} && \|\beta\|_1 \le t \end{aligned}$$

Or

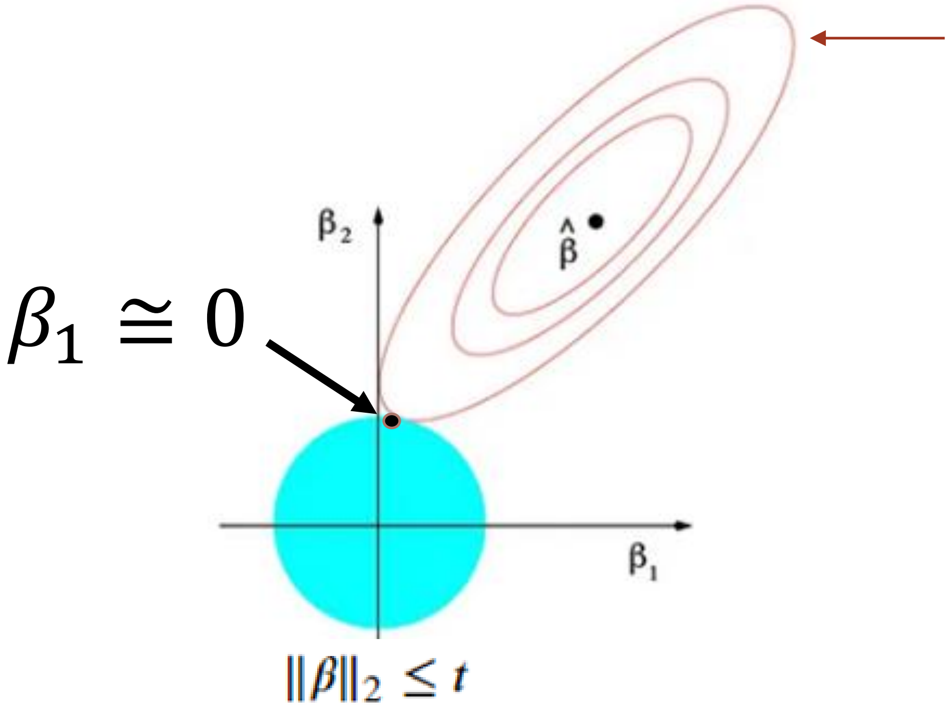
$$\underset{\beta}{\text{minimize}} \quad \frac{1}{2}\|\mathbf{y} - \mathbf{X}\beta\|_2^2 + \lambda\|\beta\|_1$$

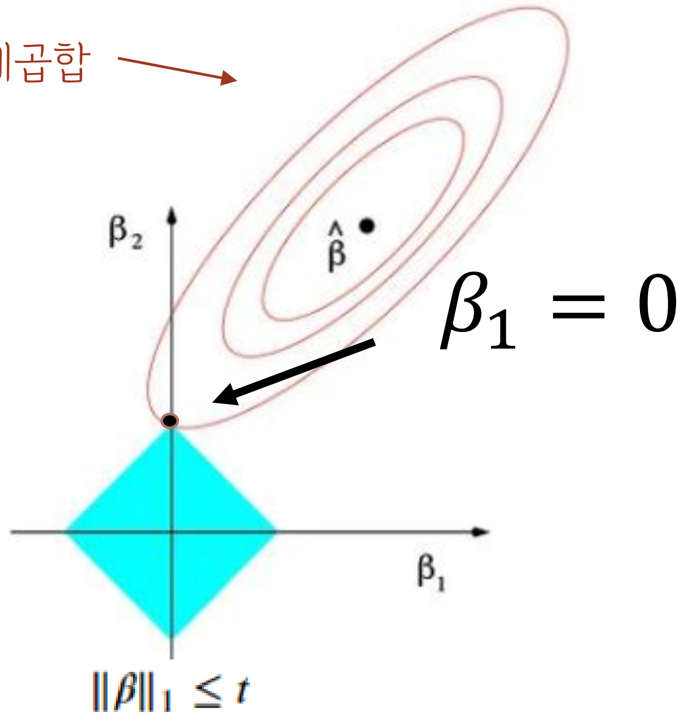
2) Lasso : β 의 l1-norm으로 제약, Sparsity

Regularization

1) Ridge

2) Lasso





∴ lasso의 제약은 중요치 않은 변수의 계수를 아예 0으로 만들어버리는 특성을 가진다!

Q1-1.

i. predictor $\mathbf{X}_i \in \mathbb{R}^{100}$일때, $\mathbf{X}_i \sim N(0, \mathbf{I}),\ i = 1, \ldots, 500$를 생성하세요.

ii. true beta coefficient를 $(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, -1, -2, -3, -4, -5, -6, -7, -8, -9, -10, 0, 0, \ldots, 0)$으로 설정하고, $y_i \sim N(\mathbf{X}_i^\top \beta, 30)$을 생성하세요.

iii. 70% - 30% 비율로 train - test 데이터를 분리하세요.

```
import numpy as np
import time
```

```
np.random.seed(1004)
Xmat = np.random.normal(size=(500,100))
beta_true = np.hstack((np.arange(1, 11),np.arange(-1, -11, -1),np.zeros(80)))
ymat = np.random.normal(loc=Xmat @ beta_true, scale=np.sqrt(30))
```

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(Xmat, ymat, test_size=0.3, random_state=1004)
```

Q1-2.

ols, ridge, lasso regression 모델을 패키지를 이용해 fitting하세요. regularization parameter의 최적값을 10-fold cross validation으로 결정하세요. estimated beta coefficient와 true beta를 비교해 보세요. prediction error를 계산해 비교하고 결과를 해석해 보세요.

Q1

```python
from sklearn.linear_model import LinearRegression, RidgeCV, LassoCV

#ols
start=time.time()
ols = LinearRegression().fit(X_train, y_train)
pred_ols = ols.predict(X_test)
runtime_ols=time.time()-start
bias_ols = np.sqrt(np.mean((beta_true - ols.coef_)**2))
mspe_ols = np.sqrt(np.mean((y_test - pred_ols)**2))


#ridge
start=time.time()
ridge = RidgeCV(cv=10).fit(X_train, y_train)
pred_ridge = ridge.predict(X_test)
runtime_ridge=time.time()-start
bias_ridge = np.sqrt(np.mean((beta_true - ridge.coef_)**2))
mspe_ridge = np.sqrt(np.mean((y_test - pred_ridge)**2))


#lasso
start=time.time()
lasso = LassoCV(cv=10).fit(X_train, y_train)
pred_lasso = lasso.predict(X_test)
runtime_lasso=time.time()-start
bias_lasso = np.sqrt(np.mean((beta_true - lasso.coef_)**2))
mspe_lasso = np.sqrt(np.mean((y_test - pred_lasso)**2))
```

```python
print(f'biases are {bias_ols}, {bias_ridge}, {bias_lasso}')
print(f'mspes are {mspe_ols}, {mspe_ridge}, {mspe_lasso}')
print(f'runtimes are {runtime_ols}, {runtime_ridge}, {runtime_lasso}')
print(sum(lasso.coef_ == 0)) # lasso is sparse
```

ols ridge lasso

```
biases are 0.3548313482155406, 0.3499495832711328, 0.18749312605915727
mspes are 6.04085741399718, 6.02228661776369, 5.315092748734784
runtimes are 0.015046119689941406, 0.06777286529541016, 0.13158082962036133
49
```

Q. 왜 Lasso의 bias와 mspe가 가장 작을까?

A. True beta 값이 대부분 0으로 구성되어 있음. Lasso는 sparsity로 인해 중요치 않은 변수에 대해 계수를 0으로 만듦.

Stochastic gradient descent

$$min_x \sum_{i=1}^{m} f_i(x)$$

Gradient descent

$$x^{(k+1)} = x^{(k)} - t_k \sum_{i=1}^{m} \nabla f_i(x^{(k)})$$

VS

Stochastic gradient descent

$$x^{(k+1)} = x^{(k)} - t_k \nabla f_i(x^{(k)})$$

즉, Stochastic gradient descent 방법은 m개의 observation 중 **랜덤하게 하나**만 뽑아 gradient 계산

연산 속도가 빠르고, 최적점에 멀리 있을 때는 수렴에 효과적

그러나, 최적점에 가까이 가면 수렴이 잘 안되는 경향을 보임 → Step size 조정 / Mini batch gradient descent

Q2

Q2.

Stochastic gradient descent를 이용해 ridge regression model을 fit하고, 패키지의 결과와 비교하세요. hyperparameter(step size, stopping criterion 등)는 알아서 설정하세요. (다만 step size의 경우, 보통 $1/k$ 등의 diminishing step size를 사용하는 것이 좋습니다.)

```python
class RidgeRegressionSGD:
    def __init__(self, learning_rate=0.01, alpha=0.01, n_iterations=10000, tol=1e-6, random_state=None):
        self.learning_rate = learning_rate
        self.alpha = alpha
        self.n_iterations = n_iterations
        self.tol = tol
        self.random_state = random_state
        self.weights = None

    def fit(self, X, y):
        if self.random_state is not None:
            np.random.seed(self.random_state)
        n_samples, n_features = X.shape
        self.weights = np.random.randn(n_features)

        prev_loss = float('inf')
        for iteration in range(self.n_iterations):
            random_index = np.random.randint(n_samples)
            x_i = X[random_index]
            y_i = y[random_index]
            gradient = 2 * x_i.reshape(-1, 1) @ (x_i.reshape(1, -1) @ self.weights - y_i) + 2 * self.alpha * self.weights
            self.weights -= self.learning_rate / np.sqrt(iteration+1) * gradient

            loss = np.mean((X @ self.weights - y) ** 2) + self.alpha * np.linalg.norm(self.weights, ord=2)**2
            if np.abs(loss - prev_loss) < self.tol:
                print(f"Converged after {iteration + 1} iterations.")
                break
            prev_loss = loss
        return self.weights

    def predict(self, X):
        return X @ self.weights
```

랜덤하게 하나의 index 선택

Step size를 잘 설정하는 것이 매우 중요!

⇨ Step size = $\frac{learning\ rate}{\sqrt{반복수+1}}$

```python
%%time
model = RidgeRegressionSGD(random_state=1004)
fit = model.fit(X_train, y_train)
```

```
Converged after 9526 iterations.
CPU times: total: 672 ms
Wall time: 635 ms
```

Q2

```
fit
```

```
array([ 1.76548694e+00,  2.01926991e+00,  2.67465609e+00,  3.80501472e+00,
        4.55390684e+00,  5.41918191e+00,  6.44366947e+00,  7.42472692e+00,
        8.73344640e+00,  9.25095191e+00, -8.68801550e-01, -2.13966019e+00,
       -2.64201611e+00, -3.28707392e+00, -4.98977378e+00, -4.77618455e+00,
       -6.87127660e+00, -6.88001165e+00, -7.98592527e+00, -9.71259361e+00,
       -4.66450002e-01, -4.58482072e-03, -5.13173856e-01, -2.92276080e-01,
        8.66270950e-01,  3.71895308e-01, -6.06337412e-01, -2.71660513e-01,
       -1.40960891e-01, -4.91642858e-01, -2.87330005e-01,  5.82758800e-01,
        5.66866517e-01, -1.59597286e-01,  1.25384489e-01, -2.20900221e-01,
        5.46964369e-01,  7.30332765e-01, -1.93977249e-01, -1.28916840e-01,
       -1.38890365e-01,  1.35241203e-01,  1.00818905e+00, -3.17924626e-01,
       -6.75443203e-01, -4.30517073e-01,  4.81848233e-01, -7.11021889e-02,
       -2.65682266e-01, -8.53421449e-02, -1.16866425e-01, -2.21964292e-01,
        4.36168336e-01,  7.63911052e-02,  3.53706618e-03,  2.36767308e-02,
       -1.06439017e-01, -4.49974972e-01, -1.57883320e-01,  5.39800005e-01,
       -5.80547876e-01,  2.05955692e-01, -4.75913103e-01, -1.12769323e-01,
       -1.91050795e-01, -5.79555341e-02, -1.11711313e-01, -3.59788005e-01,
        7.56424568e-01,  7.79749969e-02,  1.31451001e-01, -1.06976985e-01,
       -1.19626503e-01, -3.34169137e-01,  7.13334840e-01, -5.95087703e-02,
       -3.36138019e-01, -2.29694234e-01, -1.27624403e-01,  2.86127055e-01,
       -3.67767256e-02,  2.10036357e-01,  6.78461521e-01,  1.09857282e+00,
       -3.67617152e-01,  2.99418198e-01,  3.23629054e-01, -3.36775940e-02,
       -1.01940440e+00,  2.35626740e-01, -3.00064200e-01,  4.44408497e-01,
        3.45237128e-01,  3.83183217e-01,  7.78838729e-02, -2.08384901e-01,
       -6.32337285e-02, -4.17484473e-01,  1.08600806e-02, -9.64744491e-03])
```

```
bias_ridge_sgd = np.sqrt(np.mean((beta_true - fit)**2))
print(bias_ridge_sgd)
```

```
0.4483429056605018
```

```
pred = model.predict(X_test)
mspe_ridge_sgd = np.sqrt(np.mean((y_test - pred)**2))
print(mspe_ridge_sgd)
```

```
6.80654758443386
```

Coordinate descent

Convex function f를 각각의 x element 에 대해 minimize 시키면 이때의 x vector가 global minimizer가 된다는 아이디어에서 출발

Coordinate descent update: $x_i^{(k)} = \underset{x_i}{\operatorname{argmin}} f(x_1^{(k)}, x_2^{(k)}, \cdots, x_{i-1}^{(k)}, x_i, x_{i+1}^{(k-1)}, \cdots, x_p^{(k-1)})$

→ 함수 f를 최소화시키는 x_i를 찾아 $x_i^{(k)}$로 업데이트, 다른 좌표축에 대해서도 반복

Ex) Lasso: 목적함수 $\frac{1}{2}\|\mathbf{y} - \mathbf{X}\beta\|_2^2 + \lambda\|\beta\|_1$ 를

β_i에 대해 최소화시키기 위해, 편미분을 하면,

$0 = \mathbf{X}_i^\top \mathbf{X}_i \beta_i + \mathbf{X}_i^\top(\mathbf{X}_{-i}\beta_{-i} - \mathbf{y}) + \lambda s_i$ (s_i는 $|\beta_i|$의 subgradient)

$$\beta_i = S_{\lambda/\|\mathbf{X}_i\|_2^2}\left(\frac{\mathbf{X}_i^\top(\mathbf{y} - \mathbf{X}_{-i}\beta_{-i})}{\mathbf{X}_i^\top \mathbf{X}_i}\right)$$

$$[S_\lambda(y)]_i = \begin{cases} y_i - \lambda & \text{if } y_i > \lambda \\ 0 & \text{if } -\lambda \le y_i \le \lambda, \\ y_i + \lambda & \text{if } y_i < -\lambda \end{cases} \quad i = 1, \dots n$$

Soft thresholding operator

Q3

Q3.

Coordinate descent를 이용해 lasso regression model을 fit하고, 패키지의 결과와 비교하세요.

```python
class LassoCD:
    def __init__(self, n_iterations=10000, alpha=95, tol=1e-8, random_state=None):
        self.n_iterations = n_iterations
        self.tol = tol
        self.alpha = alpha
        self.random_state = random_state
        self.weights = None

    def soft_threshold(self, x, threshold):
        if x > threshold:
            return x - threshold
        elif x < -threshold:
            return x + threshold
        else:
            return 0.0

    def fit(self, X, y):
        if self.random_state is not None:
            np.random.seed(self.random_state)
        n_samples, n_features = X.shape
        self.weights = np.random.randn(n_features)

        prev_loss = float('inf')
        for iteration in range(self.n_iterations):
            for i in range(X.shape[1]):
                w = X[:, i].T @ (y - np.delete(X, i, axis=1) @ np.delete(self.weights, i)) / np.linalg.norm(X[:, i])**2
                self.weights[i] = self.soft_threshold(w, self.alpha / np.linalg.norm(X[:, i])**2)

                loss = np.mean((X @ self.weights - y) ** 2) + self.alpha * np.linalg.norm(self.weights, ord=2)**2

            if np.abs(loss - prev_loss) < self.tol:
                print(f"Converged after {iteration + 1} iterations.")
                break
            prev_loss = loss
        return self.weights

    def predict(self, X):
        return X @ self.weights
```

$$[S_\lambda(y)]_i = \begin{cases} y_i - \lambda & \text{if } y_i > \lambda \\ 0 & \text{if } -\lambda \le y_i \le \lambda\,, \quad i = 1, \dots n \\ y_i + \lambda & \text{if } y_i < -\lambda \end{cases}$$

$$\beta_i = S_{\lambda/\|\mathbf{X}_i\|_2^2}\left(\frac{\mathbf{X}_i^\top(\mathbf{y} - \mathbf{X}_{-i}\beta_{-i})}{\mathbf{X}_i^\top \mathbf{X}_i}\right)$$

```python
%%time
model = LassoCD(random_state=1004)
fit = model.fit(X_train, y_train)
```

```
Converged after 22 iterations.
CPU times: total: 297 ms
Wall time: 278 ms
```

Q3

```
fit
```

```
array([ 0.78207055,  2.03607023,  2.5443404 ,  3.74941856,
        4.78817648,  6.15260663,  6.52233452,  7.55187867,
        8.88638476,  9.88897451, -0.42337476, -2.04675424,
       -2.77618952, -3.63610311, -5.25213854, -5.30616268,
       -7.07163305, -7.34227387, -8.47257931, -10.1572751 ,
       -0.32912062,  0.        , -0.1753614 ,  0.        ,
        0.33081636,  0.        ,  0.        ,  0.        ,
        0.        ,  0.        , -0.09314883,  0.15164584,
        0.        , -0.21053926,  0.        ,  0.        ,
        0.        ,  0.31158132,  0.        ,  0.        ,
        0.        ,  0.        ,  0.13165017, -0.15622708,
        0.        ,  0.        ,  0.        , -0.106019  ,
        0.        ,  0.        ,  0.        ,  0.        ,
        0.        ,  0.        ,  0.14128109,  0.        ,
        0.        ,  0.        ,  0.        ,  0.        ,
        0.        ,  0.02815219, -0.05783665,  0.        ,
        0.        ,  0.        ,  0.        ,  0.        ,
        0.04952274,  0.        ,  0.        ,  0.        ,
        0.        , -0.06509194,  0.4966994 ,  0.        ,
        0.        ,  0.        ,  0.        , -0.07997047,
        0.        ,  0.        ,  0.        ,  0.26321596,
       -0.08250341,  0.01802504,  0.        ,  0.        ,
       -0.07715428,  0.        ,  0.        ,  0.        ,
        0.        ,  0.        ,  0.        ,  0.        ,
        0.31203752, -0.05651076,  0.03731169,  0.        ])
```

0인 값들이 보임

Lasso의 sparsity 확인

```
bias_lasso_cd = np.sqrt(np.mean((beta_true - fit)**2))
print(bias_lasso_cd)
```

0.18939876198970013

```
pred = model.predict(X_test)
mspe_lasso_cd = np.sqrt(np.mean((y_test - pred)**2))
print(mspe_lasso_cd)
```

5.385681918580375

```
sum(fit == 0)
```

56

Q4

Q4.

tensorflow 나 pytorch 패키지의 SGD, Adam optimizer를 이용해 문제를 다시 풀고 결과를 비교해 보세요. 처음 생성한 데이터를 패키지의 문법에 맞게 적절히 변형해야 할 것입니다.

```python
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
```

```python
Xtrain = torch.from_numpy(X_train.astype(np.float32))
ytrain = torch.from_numpy(y_train.astype(np.float32)).unsqueeze(dim=1)
Xtest = torch.from_numpy(X_test.astype(np.float32))
ytest = torch.from_numpy(y_test.astype(np.float32)).unsqueeze(dim=1)
```

```python
class LinearRegression(nn.Module):
    def __init__(self):
        super(LinearRegression, self).__init__()
        self.linear = nn.Linear(100, 1, bias=False)

    def forward(self, x):
        return self.linear(x)

model = LinearRegression()

criterion = nn.MSELoss()

optimizer = torch.optim.SGD(model.parameters(), lr=0.01)

start=time.time()
num_epochs = 4000
for epoch in range(num_epochs):
    outputs = model(Xtrain)
    loss = criterion(outputs, ytrain)

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    if (epoch+1) % 500 == 0:
        print('Epoch [{}/{}], Loss: {:.4f}'.format(epoch+1, num_epochs, loss.item()))

print('Learned parameters: ')
for name, param in model.named_parameters():
    if param.requires_grad:
        print(name, param.data)
        weights = param.detach().numpy()

print("\n")
print("runtime is {}".format(time.time()-start))
```

선형회귀 : nn.Linear 함수 사용

```
Epoch [500/4000], Loss: 19.9227
Epoch [1000/4000], Loss: 19.8427
Epoch [1500/4000], Loss: 19.8424
Epoch [2000/4000], Loss: 19.8424
Epoch [2500/4000], Loss: 19.8424
Epoch [3000/4000], Loss: 19.8424
Epoch [3500/4000], Loss: 19.8424
Epoch [4000/4000], Loss: 19.8424
Learned parameters:
linear.weight tensor([[  0.7989,   2.2732,   3.1168,   4.4987,   5.1974,   6.2224,   7.0459,
           7.5797,   8.9836,  10.2100,  -0.7125,  -2.7112,  -2.8725,  -3.8402,
          -5.2597,  -5.5016,  -7.4455,  -7.8027,  -9.1291, -10.8011,  -0.7537,
          -0.3833,  -0.5828,  -0.1273,   0.8474,   0.2500,  -0.5420,   0.0640,
           0.4013,  -0.2838,  -0.5139,   0.6559,   0.2448,  -0.6277,   0.2077,
           0.1538,  -0.2568,   0.7989,   0.0738,   0.2319,   0.2293,  -0.2420,
           0.2371,  -0.3416,  -0.4124,  -0.1051,   0.2767,  -0.4329,  -0.3221,
          -0.2707,   0.3150,  -0.0671,   0.1048,  -0.0135,   0.4300,  -0.0585,
          -0.1511,   0.4059,  -0.2462,  -0.1244,  -0.1862,   0.2809,  -0.2340,
          -0.2964,  -0.3448,  -0.2592,  -0.4343,  -0.2443,   0.6802,  -0.1233,
           0.1514,  -0.0691,  -0.1340,  -0.3585,   0.8627,  -0.0282,   0.0513,
           0.0563,   0.0341,  -0.4403,  -0.1384,  -0.0390,   0.1535,   0.6166,
          -0.2084,  -0.0397,   0.0972,  -0.2879,  -0.4896,  -0.1750,  -0.5980,
          -0.2136,   0.4334,  -0.1227,   0.1560,  -0.2519,   0.6577,  -0.1804,
           0.3208,  -0.2241]])


runtime is 0.8551726341247559
```

```python
prediction = model(Xtest)
bias_ols_torch = np.sqrt(np.mean((beta_true - weights.flatten())**2))
mspe_ols_torch = np.sqrt(np.mean((y_test - prediction.detach().numpy().flatten())**2))
print(f'bias: torch result {bias_ols_torch} vs sklearn result {bias_ols}')
print(f'mspe: torch result {mspe_ols_torch} vs sklearn result {mspe_ols}')
```

```
bias: torch result 0.3578517350199359 vs sklearn result 0.354831348215541
mspe: torch result 6.096740057223453 vs sklearn result 6.040857413997176
```

Q4

```python
class RidgeRegression(nn.Module):
    def __init__(self, alpha):
        super(RidgeRegression, self).__init__()
        self.linear = nn.Linear(100, 1, bias=False)
        self.alpha = alpha

    def forward(self, x):
        return self.linear(x)


alpha = ridge.alpha_
model = RidgeRegression(alpha)

criterion = nn.MSELoss()

optimizer = torch.optim.SGD(model.parameters(), lr=0.01)

start=time.time()
num_epochs = 1000
for epoch in range(num_epochs):
    outputs = model(Xtrain)
    loss = criterion(outputs, ytrain)

    ridge_loss = torch.norm(model.linear.weight, p=2)
    loss += alpha * ridge_loss

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    if (epoch+1) % 100 == 0:
        print('Epoch [{}/{}], Loss: {:.4f}'.format(epoch+1, num_epochs, loss.item()))

print('Learned parameters: ')
for name, param in model.named_parameters():
    if param.requires_grad:
        print(name, param.data)
        weights = param.detach().numpy()

print("₩n")
print("runtime is {}".format(time.time()-start))
```

Loss function에
규제항을 더해준다!

$$\frac{1}{2}\|\mathbf{y}-\mathbf{X}\beta\|_2^2+\lambda\|\beta\|_2^2$$

```
Epoch [100/1000], Loss: 75.4405
Epoch [200/1000], Loss: 52.1833
Epoch [300/1000], Loss: 48.8247
Epoch [400/1000], Loss: 48.1403
Epoch [500/1000], Loss: 47.9795
Epoch [600/1000], Loss: 47.9385
Epoch [700/1000], Loss: 47.9274
Epoch [800/1000], Loss: 47.9243
Epoch [900/1000], Loss: 47.9234
Epoch [1000/1000], Loss: 47.9232
Learned parameters:
linear.weight tensor([[ 9.1095e-01,  2.1989e+00,  2.9903e+00,  4.3094e+00,  5.0024e+00,
          6.0675e+00,  6.8453e+00,  7.4557e+00,  8.8193e+00,  1.0004e+01,
         -7.3198e-01, -2.5840e+00, -2.8343e+00, -3.7314e+00, -5.1916e+00,
         -5.3382e+00, -7.2710e+00, -7.5887e+00, -8.8838e+00, -1.0469e+01,
         -7.2762e-01, -3.2560e-01, -5.6631e-01, -9.8006e-02,  8.4113e-01,
          2.2629e-01, -5.4376e-01,  8.9429e-03,  3.2627e-01, -2.9372e-01,
         -4.6356e-01,  6.0412e-01,  2.5661e-01, -5.7766e-01,  1.9729e-01,
          1.0722e-01, -1.8155e-01,  7.4170e-01,  7.3233e-02,  1.8896e-01,
          2.1100e-01, -2.1334e-01,  3.5453e-01, -3.9392e-01, -3.9568e-01,
         -1.4145e-01,  2.9052e-01, -4.0526e-01, -3.3298e-01, -2.2285e-01,
          2.3635e-01, -7.7883e-02,  1.2022e-01, -8.7523e-03,  3.9141e-01,
         -9.7348e-02, -1.7416e-01,  2.8768e-01, -2.3780e-01, -3.5031e-02,
         -2.3092e-01,  2.8769e-01, -2.4944e-01, -2.6779e-01, -2.8515e-01,
         -2.4712e-01, -3.8897e-01, -2.5054e-01,  6.3978e-01, -5.7637e-02,
          1.4455e-01, -8.4602e-02, -1.1161e-01, -3.4648e-01,  8.6379e-01,
         -2.0938e-02, -3.5927e-02,  7.1585e-03,  6.0845e-02, -3.7790e-01,
         -1.6845e-01, -1.5161e-02,  1.9010e-01,  6.3382e-01, -1.9582e-01,
         -4.3508e-04,  1.0167e-01, -2.3170e-01, -5.3967e-01, -1.1298e-01,
         -5.4160e-01, -1.1779e-01,  3.8264e-01, -3.4028e-02,  1.5449e-01,
         -2.4621e-01,  6.1141e-01, -2.1399e-01,  2.7615e-01, -2.2918e-01]])


runtime is 0.4031195640563965
```

```python
prediction = model(Xtest)
bias_ridge_torch = np.sqrt(np.mean((beta_true - weights.flatten())**2))
mspe_ridge_torch = np.sqrt(np.mean((y_test - prediction.detach().numpy().flatten())**2))
print(f'bias: torch result {bias_ridge_torch} vs sklearn result {bias_ridge} vs our sgd result {bias_ridge_sgd}')
print(f'mspe: torch result {mspe_ridge_torch} vs sklearn result {mspe_ridge} vs our sgd result {mspe_ridge_sgd}')
```

```
bias: torch result 0.33542883941088525 vs sklearn result 0.3499495832711338 vs our sgd result 0.4483429056605018
mspe: torch result 6.014620480933848 vs sklearn result 6.022286617763701 vs our sgd result 6.80654758443386
```

Q4

```python
class LassoRegression(nn.Module):
    def __init__(self, alpha):
        super(LassoRegression, self).__init__()
        self.linear = nn.Linear(100, 1, bias=False)
        self.alpha = alpha

    def forward(self, x):
        return self.linear(x)

alpha = lasso.alpha_
model = LassoRegression(alpha)

criterion = nn.MSELoss()

optimizer = torch.optim.Adam(model.parameters(), lr=0.01)

start=time.time()
num_epochs = 4000
for epoch in range(num_epochs):
    outputs = model(Xtrain)
    loss = criterion(outputs, ytrain)

    lasso_loss = torch.sum(torch.abs(model.linear.weight)) # L1 norm of the weights
    loss += alpha * lasso_loss

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    if (epoch+1) % 500 == 0:
        print('Epoch [{}/{}], Loss: {:.4f}'.format(epoch+1, num_epochs, loss.item()))

print('Learned parameters: ')
for name, param in model.named_parameters():
    if param.requires_grad:
        print(name, param.data)
        weights = param.detach().numpy()
print("\n")
print("runtime is {}".format(time.time()-start))
```

$$\frac{1}{2}\|\mathbf{y} - \mathbf{X}\beta\|_2^2 + \lambda\|\beta\|_1$$

Q. 왜 여기선 0으로 못 만들까?

A. 토치에선 gradient기반 알고리즘을 사용하기 때문!

```
Epoch [500/4000], Loss: 199.5278
Epoch [1000/4000], Loss: 78.6239
Epoch [1500/4000], Loss: 53.5968
Epoch [2000/4000], Loss: 50.5700
Epoch [2500/4000], Loss: 50.3454
Epoch [3000/4000], Loss: 50.3373
Epoch [3500/4000], Loss: 50.3382
Epoch [4000/4000], Loss: 50.3368
Learned parameters:
linear.weight tensor([[ 7.9653e-01,  2.1490e+00,  2.7952e+00,  4.0601e+00,  5.0034e+00,
          6.1576e+00,  6.8004e+00,  7.5963e+00,  8.9825e+00,  1.0092e+01,
         -5.5646e-01, -2.3346e+00, -2.8590e+00, -3.7597e+00, -5.3267e+00,
         -5.3944e+00, -7.2455e+00, -7.5222e+00, -8.7470e+00, -1.0442e+01,
         -5.3375e-01, -6.4072e-02, -3.9284e-01, -3.1828e-02,  5.9377e-01,
          7.4576e-02, -2.8583e-01,  2.5546e-03,  1.3459e-01, -1.6375e-02,
         -2.2610e-01,  4.0789e-01,  7.5753e-02, -4.2094e-01,  7.8265e-03,
          4.7530e-03, -4.5452e-02,  5.5656e-01,  4.9989e-04,  6.0751e-02,
          1.1086e-01, -3.0716e-02,  1.8777e-01, -2.3810e-01, -1.7222e-01,
         -2.0189e-02,  3.9380e-03, -2.9037e-01, -9.9660e-02, -1.6971e-03,
          6.1552e-02, -1.6338e-03,  5.3547e-03, -7.1043e-04,  2.7533e-01,
         -2.0033e-03, -5.9360e-03,  1.5150e-01, -2.7640e-04,  1.8691e-03,
         -1.4602e-01,  1.6063e-01, -1.0831e-01, -1.3916e-01, -1.2408e-01,
         -4.3329e-03, -2.3614e-01, -1.1047e-01,  3.5427e-01, -4.1807e-04,
          5.4010e-03, -3.9507e-03, -2.2798e-02, -2.1496e-01,  6.8371e-01,
         -3.4745e-03,  8.9517e-05, -8.4626e-04,  2.2145e-03, -2.7997e-01,
         -5.6663e-02, -1.2804e-03,  5.3045e-02,  4.1743e-01, -1.8911e-01,
          1.5610e-02, -1.8070e-03, -1.3537e-03, -3.2532e-01,  7.9981e-04,
         -2.3341e-01, -2.8321e-04,  1.5738e-01,  5.8448e-03, -4.5379e-04,
         -1.2374e-01,  4.9484e-01, -1.5489e-01,  1.8116e-01, -6.9135e-02]])


runtime is 1.4069626331329346
```

```python
prediction = model(Xtest)
bias_lasso_torch = np.sqrt(np.mean((beta_true - weights.flatten())**2))
mspe_lasso_torch = np.sqrt(np.mean((y_test - prediction.detach().numpy().flatten())**2))
print(f'bias: torch result {bias_lasso_torch} vs sklearn result {bias_lasso} vs our cd result {bias_lasso_cd}')
print(f'mspe: torch result {mspe_lasso_torch} vs sklearn result {mspe_lasso} vs our cd result {mspe_lasso_cd}')
print(sum(weights.flatten() == 0))
```

```
bias: torch result 0.2303670845007709 vs sklearn result 0.18749312605915722 vs our cd result 0.18939876198970013
mspe: torch result 5.44543026442635 vs sklearn result 5.315092748734785 vs our cd result 5.385681918580375
0
```

감사합니다