Compressive sensing: tomography reconstruction with L1 prior (Lasso) [2pt]

This example shows the reconstruction of an image from a set of parallel projections, acquired along different angles. Such a dataset is acquired in **computed tomography** (CT).

Without any prior information on the sample, the number of projections required to reconstruct the image is of the order of the linear size `l` of the image (in pixels). For simplicity we consider here a sparse image, where only pixels on the boundary of objects have a non-zero value. Such data could correspond for example to a cellular material. Note however that most images are sparse in a different basis, such as the Haar wavelets. Only `l/7` projections are acquired, therefore it is necessary to use prior information available on the sample (its sparsity): this is an example of **compressive sensing**.

The tomography projection operation is a linear transformation. In addition to the data-fidelity term corresponding to a linear regression, we penalize the L1 norm of the image to account for its sparsity. The resulting optimization problem is called the `lasso`. We use the class :class:`~sklearn.linear_model.Lasso`, that uses the coordinate descent algorithm. Importantly, this implementation is more computationally efficient on a sparse matrix, than the projection operator used here.

The reconstruction with L1 penalization gives a result with zero error (all pixels are successfully labeled with 0 or 1), even if noise was added to the projections. In comparison, an L2 penalization (:class:`~sklearn.linear_model.Ridge`) produces a large number of labeling errors for the pixels. Important artifacts are observed on the reconstructed image, contrary to the L1 penalization. Note in particular the circular artifact separating the pixels in the corners, that have contributed to fewer projections than the central disk.

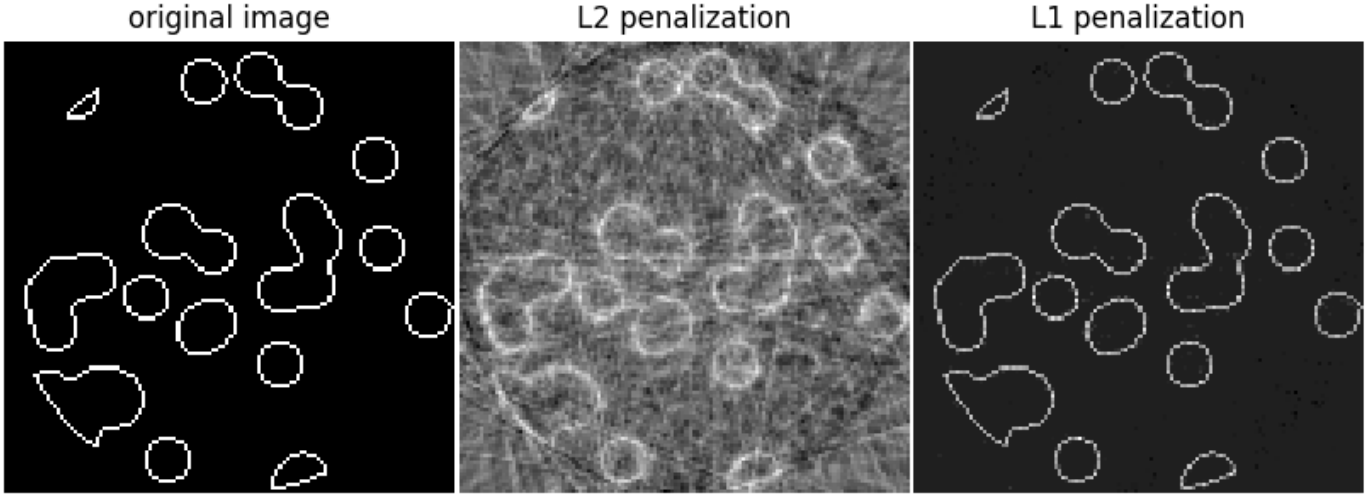
```
print(__doc__)

# Author: Emmanuelle Gouillart <emmanuelle.gouillart@nsup.org>
# License: BSD 3 clause

import numpy as np
from scipy import sparse
from scipy import ndimage
```

```
from sklearn.linear_model import Lasso
from sklearn.linear_model import Ridge
import matplotlib.pyplot as plt


def _weights(x, dx=1, orig=0):
    x = np.ravel(x)
    floor_x = np.floor((x - orig) / dx).astype(np.int64)
    alpha = (x - orig - floor_x * dx) / dx
    return np.hstack((floor_x, floor_x + 1)), np.hstack((1 - alpha, alpha))


def _generate_center_coordinates(l_x):
    X, Y = np.mgrid[:l_x, :l_x].astype(np.float64)
    center = l_x / 2.
    X += 0.5 - center
    Y += 0.5 - center
    return X, Y


def build_projection_operator(l_x, n_dir):
    """ Compute the tomography design matrix.

    Parameters
    ----------

    l_x : int
        linear size of image array

    n_dir : int
        number of angles at which projections are acquired.

    Returns
    -------
    p : sparse matrix of shape (n_dir l_x, l_x**2)
    """
    X, Y = _generate_center_coordinates(l_x)
    angles = np.linspace(0, np.pi, n_dir, endpoint=False)
    data_inds, weights, camera_inds = [], [], []
    data_unravel_indices = np.arange(l_x ** 2)
    data_unravel_indices = np.hstack((data_unravel_indices,
                                      data_unravel_indices))
    for i, angle in enumerate(angles):
        Xrot = np.cos(angle) * X - np.sin(angle) * Y
        inds, w = _weights(Xrot, dx=1, orig=X.min())
```

```
            mask = np.logical_and(inds >= 0, inds < l_x)
            weights += list(w[mask])
            camera_inds += list(inds[mask] + i * l_x)
            data_inds += list(data_unravel_indices[mask])
        proj_operator = sparse.coo_matrix((weights, (camera_inds, data_inds)))
        return proj_operator


    def generate_synthetic_data():
        """ Synthetic binary data """
        rs = np.random.RandomState(0)
        n_pts = 36
        x, y = np.ogrid[0:l, 0:l]
        mask_outer = (x - l / 2.) ** 2 + (y - l / 2.) ** 2 < (l / 2.) ** 2
        mask = np.zeros((l, l))
        points = l * rs.rand(2, n_pts)
        mask[(points[0]).astype(int), (points[1]).astype(int)] = 1
        mask = ndimage.gaussian_filter(mask, sigma=l / n_pts)
        res = np.logical_and(mask > mask.mean(), mask_outer)
        return np.logical_xor(res, ndimage.binary_erosion(res))


    # Generate synthetic images, and projections
    l = 128
    proj_operator = build_projection_operator(l, l // 7)
    data = generate_synthetic_data()
    proj = proj_operator @ data.ravel()[:, np.newaxis]
    proj += 0.15 * np.random.randn(*proj.shape)

    # Reconstruction with L2 (Ridge) penalization
    rgr_ridge = Ridge(alpha=0.2)
    rgr_ridge.fit(proj_operator, proj.ravel())
    rec_l2 = rgr_ridge.coef_.reshape(l, l)

    # Reconstruction with L1 (Lasso) penalization
    # the best value of alpha was determined using cross validation
    # with LassoCV
    rgr_lasso = Lasso(alpha=0.001)
    rgr_lasso.fit(proj_operator, proj.ravel())
    rec_l1 = rgr_lasso.coef_.reshape(l, l)

    plt.figure(figsize=(8, 3.3))
    plt.subplot(131)
    plt.imshow(data, cmap=plt.cm.gray, interpolation='nearest')
    plt.axis('off')
```

```
plt.title('original image')
plt.subplot(132)
plt.imshow(rec_l2, cmap=plt.cm.gray, interpolation='nearest')
plt.title('L2 penalization')
plt.axis('off')
plt.subplot(133)
plt.imshow(rec_l1, cmap=plt.cm.gray, interpolation='nearest')
plt.title('L1 penalization')
plt.axis('off')

plt.subplots_adjust(hspace=0.01, wspace=0.01, top=1, bottom=0, left=0,
                    right=1)

plt.show()
```

```
Automatically created module for IPython interactive environment
```



Question1

```
from sklearn.linear_model import LassoCV
import numpy as np


alpha_values=np.array([0.001,0.005,0.01,0.05,0.1])
```

```
lassoreg_crossvalid = LassoCV(cv=3, random_state=0, alphas=alpha_values)
lassoreg_crossvalid.fit(proj_operator,proj.ravel())
```

```
LassoCV
LassoCV(alphas=array([0.001, 0.005, 0.01 , 0.05 , 0.1  ]), cv=3, random_state=
```

```
print("Intercept ", lassoreg_crossvalid.intercept_)
print("Coefficients ", lassoreg_crossvalid.coef_)
print("The Best Alpha value is ", lassoreg_crossvalid.alpha_)
```

```
Intercept  0.5960845169538995
Coefficients  [-0. -0. -0. ...  0.  0.  0.]
The Best Alpha value is  0.001
```

In this homework, there are three different datasets consisting of 2-dimensional input features and binary class labels, and you will be asked to implement machine learning classifiers. Total 3 pts.

Let's begin by importing some libaries.

```
import sys; sys.path.append('../..') ; sys.path.append('..') ; from my_utils impo

import torch
import torch.nn as nn
import torch.utils.data as data
# dummy trainloader
trainloader = data.DataLoader(data.TensorDataset(torch.Tensor(1), torch.Tensor(1)
device = torch.device('cpu')

import matplotlib.pyplot as plt
```

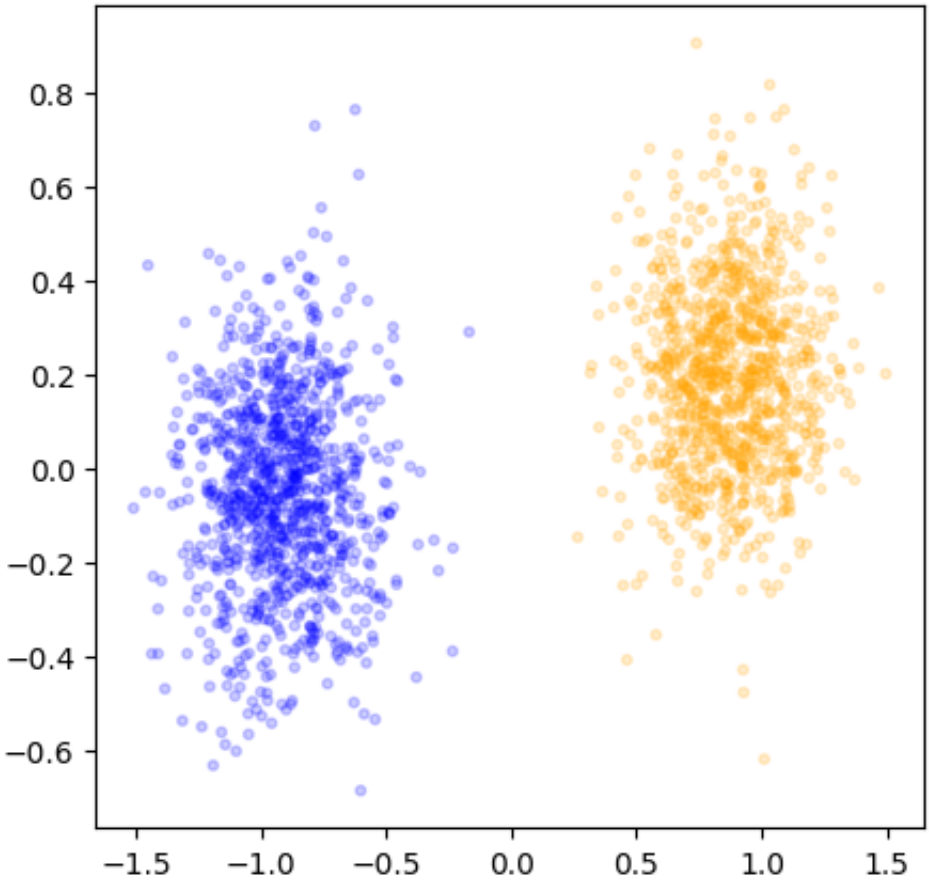
Next, we set a random seed for reproducibility.

```
import numpy as np
import random

seed = 0
np.random.seed(seed)
torch.random.manual_seed(seed)
random.seed(seed)
```

Two Gaussian blobs

This is the first dataset, Gaussian distributed.

```
X, y = sample_gaussian()
fig, ax = plt.subplots(1,1, figsize=(5,5))
plot_scatter(ax, X, y)
```



Your task is to build a binary classifier based on logistic regression.

[1 pt] Fill in the following class template to perform logistic regression.

```
class Model(nn.Module):
    def __init__(self,device="cpu"):
        super(Model, self).__init__()
        self.linear = torch.nn.Linear(2,1)

    def forward(self, x):
      #Finding the prediction
      y_pred = self.linear(x)
      y = torch.sigmoid(y_pred)
      return y
```

```
model = Model().to(device)
```

We will be using AdamW optimizer.

```
import torch.optim as optim
optimizer = optim.AdamW(model.parameters(), lr=1e-2, weight_decay=1e-6)

loss_function = torch.nn.BCELoss() # Binary Cross Entropy to calculate the loss
```

With the defined model (logistic_reg) and the optimizer, we will train the model using binary cross entropy.

[1 pt] Finish implementing the training loop.

```
for itr in range(1, 200001):
    optimizer.zero_grad()
    yh = torch.squeeze(model(X)) # forward pass
    loss = loss_function(yh.float(),y.float()) # compute loss
    loss.backward() # backward pass
    if itr%50000 == 0:
      print("Number of Iteration ", itr, " Loss is ", loss)
    optimizer.step()
```

```
Number of Iteration  50000  Loss is  tensor(1.0289e-09, grad_fn=<BinaryCrossEr
Number of Iteration  100000  Loss is  tensor(2.2960e-10, grad_fn=<BinaryCrossE
Number of Iteration  150000  Loss is  tensor(1.1665e-10, grad_fn=<BinaryCrossE
Number of Iteration  200000  Loss is  tensor(1.1225e-10, grad_fn=<BinaryCrossE
```

```
for name, parameter in model.named_parameters():
  print(name, parameter)
```

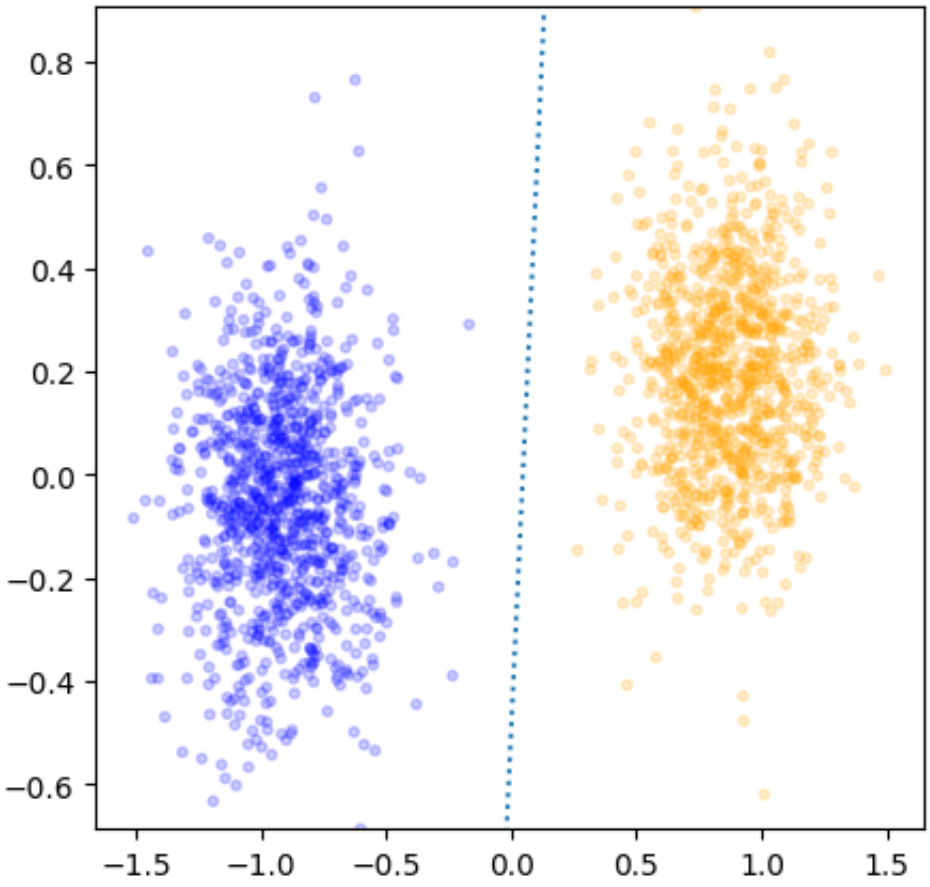
```
linear.weight Parameter containing:
tensor([[69.2933, -6.5525]], requires_grad=True)
linear.bias Parameter containing:
tensor([-3.2043], requires_grad=True)
```

With the trained model, make predictions on training set.

[1 pt] Draw a plot depicting the data points that are color coded based on the predicted labels, and the decision boundary leared by the logistic regression. See the example below.

```
p=list(model.parameters())
weight=p[0][0].detach().numpy()
bias=p[1].detach().numpy()
x_line=np.linspace(X[:, 0].min(), X[:, 0].max(), 2)
db=-(bias + weight[0]*x_line)/weight[1]
```

```
with torch.no_grad():
    y_pred=model(X)
    y_class=y_pred.round()
    fig,ax = plt.subplots(1,1,figsize=(5,5))
    plot_scatter(ax,X,y)
    plt.ylim(X[:, 1].min(), X[:, 1].max())
    plt.plot(x_line,db,linestyle='dotted')
```



3) $l(\theta) = y \log(h(\theta^T x)) + (1-y)\log(1-h(\theta^T x))$

$h(z) = \frac{1}{1+e^{-z}} \qquad \theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \end{bmatrix} \qquad x = \begin{bmatrix} 1 \\ x_1 \\ x_2 \end{bmatrix}$

$h(\theta^T x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2$

$\nabla_\theta l(\theta) = \begin{bmatrix} \frac{\partial l(\theta)}{\partial \theta_0} \\ \frac{\partial l(\theta)}{\partial \theta_1} \\ \frac{\partial l(\theta)}{\partial \theta_2} \end{bmatrix}$

$\frac{\partial l(\theta)}{\partial \theta_0} = \frac{y}{h(\theta^T x)} \times h(\theta^T x)(1-h(\theta^T x)) \times 1 + \frac{(1-y)}{(1-h(\theta^T x))} \times (-1)(1-h(\theta^T x))(h(\theta^T x))$

$= y(1-h(\theta^T x)) + (y-1)(h(\theta^T x))$

$y - y h(\theta^T x) + y h(\theta^T x) - h(\theta^T x)$

$y - h(\theta^T x)$

Similarly,

$\frac{\partial l(\theta)}{\partial \theta_1} = (y - h(\theta^T x)) x_1$

$\frac{\partial l(\theta)}{\partial \theta_2} = (y - h(\theta^T x)) x_2$

$\therefore \quad \nabla_\theta l(\theta) = \begin{bmatrix} y - h(\theta^T x) \\ (y - h(\theta^T x)) x_1 \\ (y - h(\theta^T x)) x_2 \end{bmatrix}$