Name - Suhas Raghavendra

ASU ID - 1233271600 ¶

Following is the code taken from the scikit-learn website (https://scikit-learn.org/stable/auto_examples/applications/plot_tomography_l1_reconstruction.html (https://scikit-learn.org/stable/auto_examples/applications/plot_tomography_l1_reconstruction.html))

[2pt] The author claims that the alpha value (0.001) is chosen by running the cross validation using LassoCV. Your task is to validate the claim by writing the code for performing LassoCV (from scikit-learn). Construct a set containing at least 5 candiate alpha values (of course, including 0.001) and perform LassoCV. Check whether LassoCV gives 0.001.

In [1]:
```
%matplotlib inline
```

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.

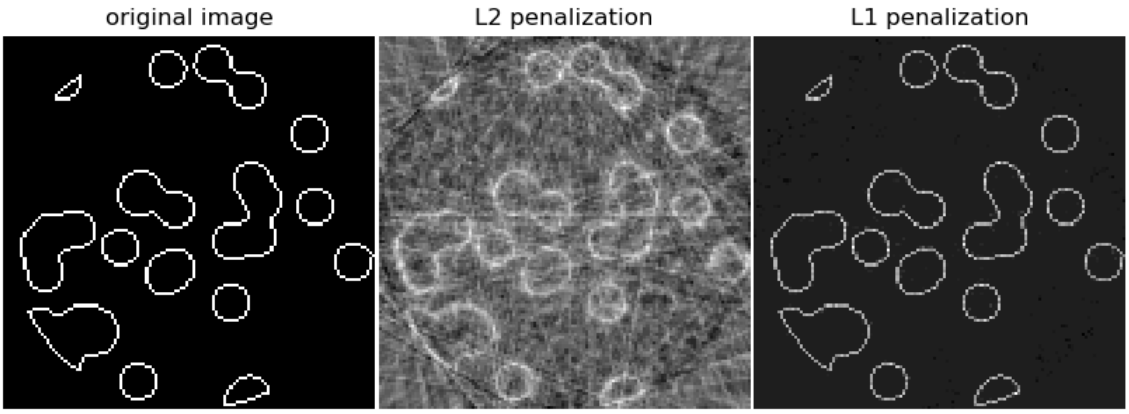
The reconstruction with L1 penalization gives a result with zero error (all pixels are successfully labeled with 0 or 1), even if noise was added to the projections. In comparison, an L2 penalization (:class: `~sklearn.linear_model.Ridge`) produces a large number of labeling errors for the pixels. Important 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.

In [2]:

```
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

#importing lassoCV module
from sklearn.linear_model import LassoCV, Ridge

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()
```

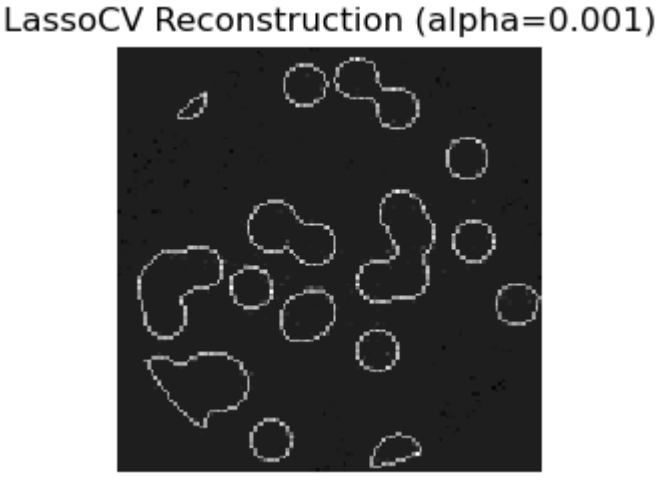
```
Automatically created module for IPython interactive environment
```



In [3]:
```
alphas = [0.001, 0.01, 0.1, 0.2, 0.3, 0.05]

#running the LassoCV regression we imported before
lasso_cv_validation = LassoCV(alphas=alphas, cv=5, max_iter=10000)
lasso_cv_validation.fit(proj_operator, proj.ravel())

best_alpha = lasso_cv_validation.alpha_
print("best alpha = " + str(best_alpha))

rec_lasso_cv = lasso_cv_validation.coef_.reshape(l, l)
```

```
best alpha = 0.001
```

In [4]:
```
plt.subplot(132)
plt.imshow(rec_lasso_cv, cmap=plt.cm.gray, interpolation="nearest")
plt.title(f"LassoCV Reconstruction (alpha={best_alpha})")
plt.axis("off")
plt.subplots_adjust(hspace=0.01, wspace=0.01, top=1, bottom=0, left=0, right
plt.show()
```



In [ ]:

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.

In [2]:

```
import math

import numpy as np
import torch
from torch import sqrt, pow, cat, zeros, Tensor
from scipy.integrate import solve_ivp
from sklearn.neighbors import KernelDensity
from torch.distributions import Normal

class ToyDataset:
    """Handles the generation of classification toy datasets"""
    def generate(self, n_samples:int, dataset_type:str, **kwargs):
        """Handles the generation of classification toy datasets
        :param n_samples: number of datasets points in the generated dataset
        :type n_samples: int
        :param dataset_type: {'moons', 'spirals', 'spheres', 'gaussians', 'g
        :type dataset_type: str
        :param dim: if 'spheres': dimension of the spheres
        :type dim: float
        :param inner_radius: if 'spheres': radius of the inner sphere
        :type inner_radius: float
        :param outer_radius: if 'spheres': radius of the outer sphere
        :type outer_radius: float
        """
        if dataset_type == 'moons':
            return generate_moons(n_samples=n_samples, **kwargs)
        elif dataset_type == 'spirals':
            return generate_spirals(n_samples=n_samples, **kwargs)
        elif dataset_type == 'spheres':
            return generate_concentric_spheres(n_samples=n_samples, **kwargs
        elif dataset_type == 'gaussians':
            return generate_gaussians(n_samples=n_samples, **kwargs)
        elif dataset_type == 'gaussians_spiral':
            return generate_gaussians_spiral(n_samples=n_samples, **kwargs)
        elif dataset_type == 'diffeqml':
            return generate_diffeqml(n_samples=n_samples, **kwargs)

def randnsphere(dim:int, radius:float) -> Tensor:
    """Uniform sampling on a sphere of `dim` and `radius`

    :param dim: dimension of the sphere
    :type dim: int
    :param radius: radius of the sphere
    :type radius: float
    """
    v = torch.randn(dim)
    inv_len = radius / sqrt(pow(v, 2).sum())
    return v * inv_len

def generate_gaussians(n_samples=100, n_gaussians=7, dim=2,
                       radius=0.5, std_gaussians=0.1, noise=1e-3):
    """Creates `dim`-dimensional `n_gaussians` on a ring of radius `radius`.

    :param n_samples: number of datasets points in the generated dataset
    :type n_samples: int
    :param n_gaussians: number of gaussians distributions placed on the circ
    :type n_gaussians: int
    :param dim: dimension of the dataset. The distributions are placed on th
    :type dim: int
    :param radius: radius of the circle on which the distributions lie
    :type radius: int
```

```
    :param std_gaussians: standard deviation of the gaussians.
    :type std_gaussians: int
    :param noise: standard deviation of noise magnitude added to each datase
    :type noise: float
    """
    X = torch.zeros(n_samples * n_gaussians, dim) ; y = torch.zeros(n_sample
    angle = torch.zeros(1)
    if dim > 2: loc = torch.cat([radius*torch.cos(angle), radius*torch.sin(a
    else: loc = torch.cat([radius*torch.cos(angle), radius*torch.sin(angle)]
    dist = Normal(loc, scale=std_gaussians)

    for i in range(n_gaussians):
        angle += 2*math.pi / n_gaussians
        if dim > 2: dist.loc = torch.Tensor([radius*torch.cos(angle), torch
        else: dist.loc = torch.Tensor([radius*torch.cos(angle), radius*torch
        X[i*n_samples:(i+1)*n_samples] = dist.sample(sample_shape=(n_samples
        y[i*n_samples:(i+1)*n_samples] = i
    return X, y

def generate_concentric_spheres(n_samples:int=100, noise:float=1e-4, dim:int
                                inner_radius:float=0.5, outer_radius:int=1)
    """Creates a *concentric spheres* dataset of `n_samples` datasets points

    :param n_samples: number of datasets points in the generated dataset
    :type n_samples: int
    :param noise: standard deviation of noise magnitude added to each datase
    :type noise: float
    :param dim: dimension of the spheres
    :type dim: float
    :param inner_radius: radius of the inner sphere
    :type inner_radius: float
    :param outer_radius: radius of the outer sphere
    :type outer_radius: float
    """
    X, y = zeros((n_samples, dim)), torch.zeros(n_samples)
    y[:n_samples // 2] = 1
    samples = []
    for i in range(n_samples // 2):
        samples.append(randnsphere(dim, inner_radius)[None, :])
    X[:n_samples // 2] = cat(samples)
    X[:n_samples // 2] += zeros((n_samples // 2, dim)).normal_(0, std=noise)
    samples = []
    for i in range(n_samples // 2):
        samples.append(randnsphere(dim, outer_radius)[None, :])
    X[n_samples // 2:] = cat(samples)
    X[n_samples // 2:] += zeros((n_samples // 2, dim)).normal_(0, std=noise)
    return X, y


def generate_spirals(n_samples=100, noise=1e-4, **kwargs):
    """Creates a *spirals* dataset of `n_samples` datasets points.

    :param n_samples: number of datasets points in the generated dataset
    :type n_samples: int
    :param noise: standard deviation of noise magnitude added to each datase
    :type noise: float
    """
    n = np.sqrt(np.random.rand(n_samples, 1)) * 780 * (2 * np.pi) / 360
    d1x = -np.cos(n) * n + np.random.rand(n_samples, 1) * noise
    d1y = np.sin(n) * n + np.random.rand(n_samples, 1) * noise
    X, y = (np.vstack((np.hstack((d1x, d1y)), np.hstack((-d1x, -d1y)))),
```

```
            np.hstack((np.zeros(n_samples), np.ones(n_samples))))
    X, y = torch.Tensor(X), torch.Tensor(y).long()
    return X, y

def sample_gaussian(n_samples=1<<10, device=torch.device('cpu')):
    X, y = ToyDataset().generate(n_samples, 'gaussians', n_gaussians=2, dim=
    return 2*X.to(device), y.long().to(device)

def sample_annuli(n_samples=1<<10, device=torch.device('cpu')):
    X, y = ToyDataset().generate(n_samples, 'spheres', dim=2, noise=.05)
    return 2*X.to(device), y.long().to(device)

def sample_spiral(n_samples=1<<10, device=torch.device('cpu')):
    X, y = ToyDataset().generate(n_samples, 'spirals', n_gaussians=2, dim=2,
    return 2*X.to(device), y.long().to(device)

def plot_scatter(ax, X, y):
    colors = ['blue', 'orange']
    ax.scatter(X[:,0], X[:,1], c=[colors[int(yi)] for yi in y], alpha=0.2, s
```

In [3]:
```
pip install torchvision
```

```
Requirement already satisfied: torchvision in c:\users\sos\anaconda3\lib\s
ite-packages (0.19.1)
Requirement already satisfied: numpy in c:\users\sos\anaconda3\lib\site-pa
ckages (from torchvision) (1.26.4)
Requirement already satisfied: torch==2.4.1 in c:\users\sos\anaconda3\lib
\site-packages (from torchvision) (2.4.1)
Requirement already satisfied: pillow!=8.3.*,>=5.3.0 in c:\users\sos\anaco
nda3\lib\site-packages (from torchvision) (10.3.0)
Requirement already satisfied: filelock in c:\users\sos\anaconda3\lib\site
-packages (from torch==2.4.1->torchvision) (3.13.1)
Requirement already satisfied: typing-extensions>=4.8.0 in c:\users\sos\an
aconda3\lib\site-packages (from torch==2.4.1->torchvision) (4.11.0)
Requirement already satisfied: sympy in c:\users\sos\anaconda3\lib\site-pa
ckages (from torch==2.4.1->torchvision) (1.12)
Requirement already satisfied: networkx in c:\users\sos\anaconda3\lib\site
-packages (from torch==2.4.1->torchvision) (3.2.1)
Requirement already satisfied: jinja2 in c:\users\sos\anaconda3\lib\site-p
ackages (from torch==2.4.1->torchvision) (3.1.4)
Requirement already satisfied: fsspec in c:\users\sos\anaconda3\lib\site-p
ackages (from torch==2.4.1->torchvision) (2024.3.1)
Requirement already satisfied: setuptools in c:\users\sos\anaconda3\lib\si
te-packages (from torch==2.4.1->torchvision) (69.5.1)
Requirement already satisfied: MarkupSafe>=2.0 in c:\users\sos\anaconda3\l
ib\site-packages (from jinja2->torch==2.4.1->torchvision) (2.1.3)
Requirement already satisfied: mpmath>=0.19 in c:\users\sos\anaconda3\lib
\site-packages (from sympy->torch==2.4.1->torchvision) (1.3.0)
Note: you may need to restart the kernel to use updated packages.
```

In [4]:
```
import sys; sys.path.append('../..') ; sys.path.append('..') ; from my_utils

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

import matplotlib.pyplot as plt
```

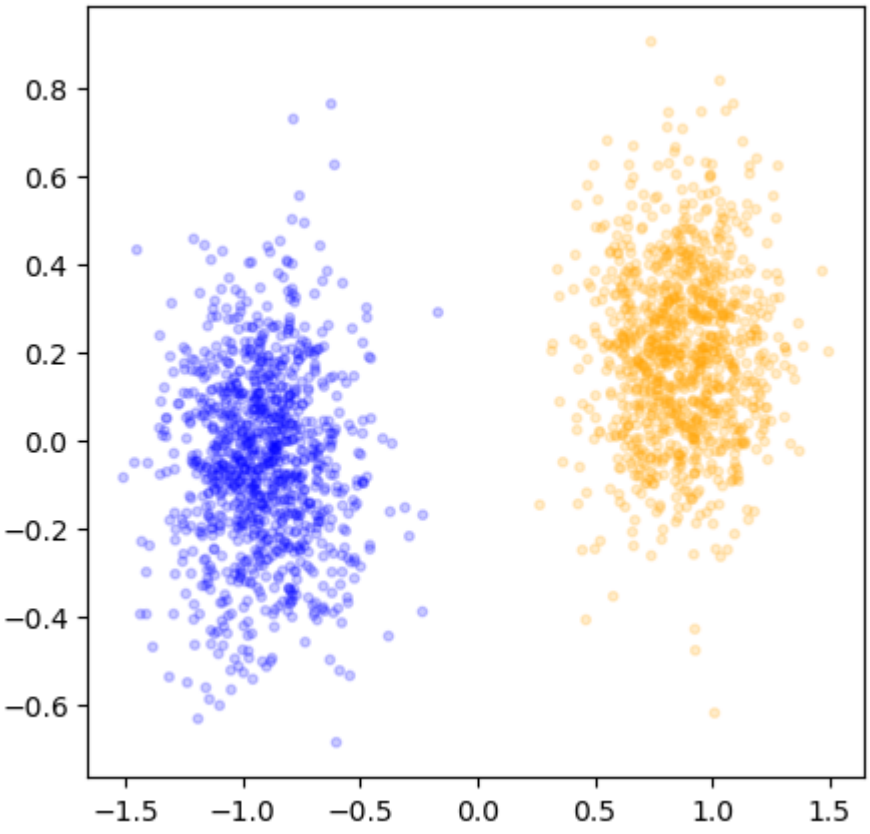
Next, we set a random seed for reproducibility.

In [6]:
```
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.

In [8]:
```
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.

In [10]:
```
class Model(nn.Module):
    def __init__(self,device="cpu"):
        super(Model, self).__init__()
        input_dim = X.shape[1]
        self.linear = nn.Linear(input_dim, 1)
        self.device = device
        self.to(device)

    def forward(self, x):
        logits = self.linear(x)
        y = torch.sigmoid(logits)

        return y
```

In [11]:
```
model = Model().to(device)
```

We will be using AdamW optimizer.

In [13]:
```
import torch.optim as optim
optimizer = optim.AdamW(model.parameters(), lr=1e-2, weight_decay=1e-6)

criterion = nn.BCELoss()
```

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.

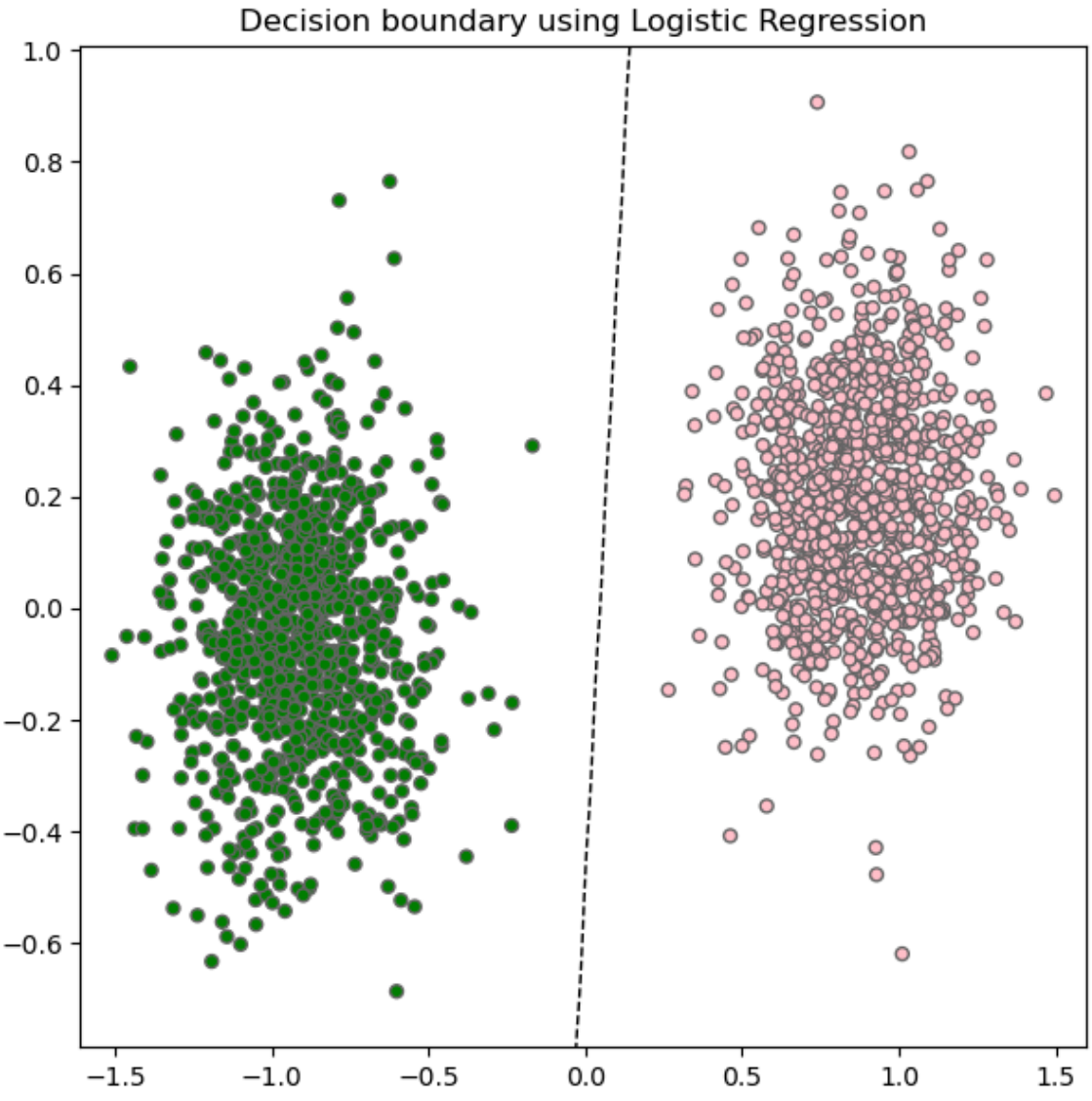
In [15]:
```
for itr in range(1, 200001):
    model.train()
    optimizer.zero_grad()
    y_prediction = model(X).squeeze() # forward pass
    y = y.float()
    loss = criterion(y_prediction, y)
    loss.backward() # backward pass
    optimizer.step()
```

```
1840 0.008382368832826614
1841 0.008375433273613453
1842 0.008368505164980888
1843 0.00836158636957407
1844 0.008354675956070423
1845 0.008347775787115097
1846 0.008340881206095219
1847 0.00833399873226881
1848 0.008327124640345573
1849 0.008320257067680359
1850 0.008313394151628017
1851 0.00830654427409172
1852 0.008299700915813446
1853 0.008292867802083492
1854 0.00828604307025671
1855 0.008279228582978249
1856 0.00827242061495781
1857 0.008265621028840542
1858 0.008258831687271595
```

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.

In [17]:
```
from matplotlib.colors import ListedColormap

with torch.no_grad():
    model.eval()
    y_prediction = model(X).squeeze()
    predicted_labels = (y_prediction >= 0.5).float()

    fig, ax = plt.subplots(figsize=(7, 7))
    colourInput = ListedColormap(['green', 'pink'])

    ax.scatter(X[:, 0], X[:, 1], c=predicted_labels, cmap=colourInput, edgec
    x_min, x_max = X[:, 0].min() - 0.1, X[:, 0].max() + 0.1
    y_min, y_max = X[:, 1].min() - 0.1, X[:, 1].max() + 0.1
    xx, yy = np.meshgrid(np.linspace(x_min, x_max, 100), np.linspace(y_min,

    grid = np.c_[xx.ravel(), yy.ravel()]
    grid_tensor = torch.Tensor(grid)
    zz = model(grid_tensor).detach().cpu().numpy().reshape(xx.shape)
    ax.contour(xx, yy, zz, levels=[0.5], linewidths=1, colors='black' ,lines

    plt.title("Decision boundary using Logistic Regression")
    plt.show()

print(y_prediction)
```



```
tensor([1.4537e-35, 1.0508e-30, 9.9325e-25,  ..., 1.0000e+00, 1.0000e+00,
        1.0000e+00])
```