In [1]:

```
'''
Dependencies
'''
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
```

In [2]:

```
'''
Reading the MNIST/USPS Handwritten Digits Dataset
'''


def readData(fname='ZipDigits.train'):
    '''
    Input:
        fname: name of file containing N examples, each
with d attributes
    Output:
        X: N x d+1 numpy array
        y: N x 1 numpy array
    '''
    with open(fname) as f:
        X = []
        y = []
        rlines = f.readlines()
        for line in rlines:
            row = line.rstrip().split(' ')
            yval = int(float(row[0]))
            y.append(yval)
            xvals = [float(pixel) for pixel in row[1:]]
            X.append(xvals)
        X = np.array(X)
        y = np.array(y)
        y = y.reshape((y.shape[0], 1))
        print(f'X shape: {X.shape}')
        print(f'y shape: {y.shape}')
        return X, y
```

In [3]:

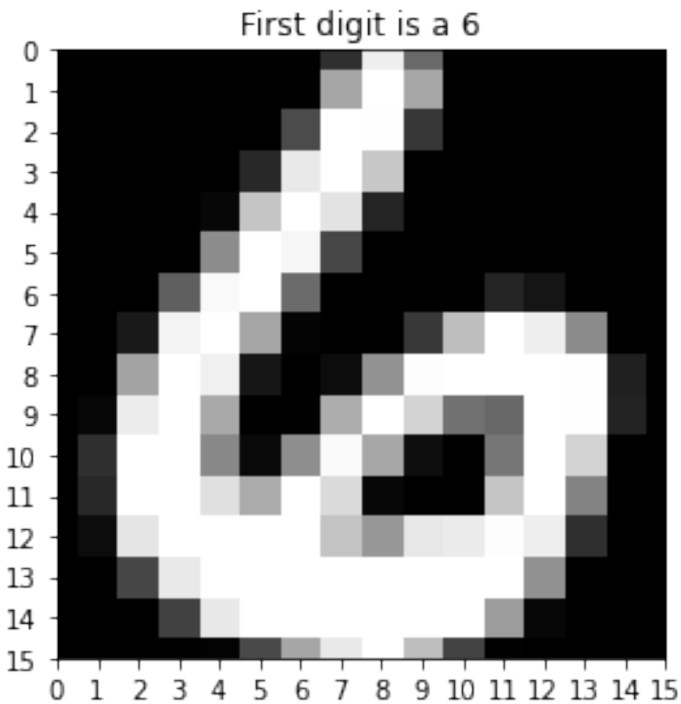
```
'''
Read training and test datasets
```

```python
'''

Xdigitstrain, ydigitstrain =
readData('ZipDigits.train')

Ndigitstrain, pixels = Xdigitstrain.shape

assert(Ndigitstrain == ydigitstrain.shape[0])

Xdigitstest, ydigitstest = readData('ZipDigits.test')

Ndigitstest, pixels = Xdigitstest.shape

assert(Ndigitstest == ydigitstest.shape[0])
```

```
X shape: (7291, 256)
y shape: (7291, 1)
X shape: (2007, 256)
y shape: (2007, 1)
```

In [4]:

```python
'''
Show images of handwritten digits
'''
def showKthImage(X, y, k):
    image = X[k, :].reshape((16, 16))
    plt.imshow(image, cmap='gray', vmin=-1, vmax=1)
    plt.title(f'First digit is a {y[k, 0]}')
    plt.xlim(0, 15)
    plt.ylim(15, 0)
    plt.xticks(range(16))
    plt.yticks(range(16))
    plt.tight_layout()
    plt.show()

showKthImage(Xdigitstrain, ydigitstrain, 0)
```



In [5]:

```
'''
Compute the augmented matrix with features
Helper Functions
'''

def computeIntensity(X):
    '''
    Input:
        X: a 2 dimensional N x 256 numpy array
           each row contains the values of 256 pixels
from a 16 x 16 grayscale image of a handwritten digit
           each pixel has an intensity value between -1
and 1
    Output:
        intensities: a 2 dimensional N x 1 numpy array
                     each row consists of a single
value representing the
                     average pixel intesity of the
corresponding image
                     See LFD Example 3.1
    '''
    print('computing intensity feature')
```

```
    N, d = X.shape
    print(f'Input shape {N}, {d}')
    '''
    TODO: Compute the intensity feature for N data
points
    '''
    print(f'Output shape {intensities.shape}')
    return intensities


def computeSymmetry(X):
    '''
    Input:
        X: a 2 dimensional N x 256 numpy array
           each row contains the values of 256 pixels
from a 16 x 16 grayscale image of a handwritten digit
           each pixel has an intensity value between -1
and 1
    Output:
        symmetries: a 2 dimensional N x 1 numpy array
                     each row consists of a single
value representing the
                     "horizontal" symmetry of the 16 x
16 image about the vertical axis
                     See LFD Example 3.1
    '''
    print('computing symmetry feature')
    N, d = X.shape
    print(f'Input shape {N}, {d}')
    Ximgs = [X[n, :].reshape((16, 16)) for n in
range(N)]
    Ximgs_flipped = [np.flip(Ximgs[n], axis=1) for n in
range(N)]
    '''
    TODO: Compute the symmetry feature for N data
points
    '''
    symmetries = symmetries.reshape(N, 1)
    print(f'Output shape {symmetries.shape}')
    return symmetries
```

```
def computeAugmentedXWithFeatures(X):
    '''
    Input:
        X: a 2 dimensional N x 256 numpy array
           each row contains the values of 256 pixels
from a 16 x 16 grayscale image of a handwritten digit
           each pixel has an intensity value between -1
and 1
    Output:
        Xaug: a 2 dimensional N x 3 numpy array
              the augmented feature matrix
              the i-th row corresponds to the i-th row
of X (and image represented by it)
              the 0-th column is the column of 1s
              the 1-st column is the column of average
intensities
              the 2-nd column is the column of
horizontal symmetries
    '''
    N, d = X.shape
    intensity = computeIntensity(X)
    symmetry = computeSymmetry(X)
    dummy = np.ones((N, 1))
    Xaug = np.concatenate((dummy, intensity, symmetry),
axis=1)
    # print(Xaug)
    print (f'Shape of augmented feature matrix:
{Xaug.shape}')
    return Xaug
```

In [6]:

```
'''
Compute the augmented matrix with features
'''

print('Computing augmented training feature matrix')

Xaugtrain = computeAugmentedXWithFeatures(Xdigitstrain)
```

```
Naugtrain, d = Xaugtrain.shape

print('Computing augmented test feature matrix')

Xaugtest = computeAugmentedXWithFeatures(Xdigitstest)

Naugtest, d = Xaugtest.shape
```

```
Computing augmented training feature matrix
computing intensity feature
Input shape 7291, 256
Output shape (7291, 1)
computing symmetry feature
Input shape 7291, 256
Output shape (7291, 1)
Shape of augmented feature matrix: (7291, 3)
Computing augmented test feature matrix
computing intensity feature
Input shape 2007, 256
Output shape (2007, 1)
computing symmetry feature
Input shape 2007, 256
Output shape (2007, 1)
Shape of augmented feature matrix: (2007, 3)
```

In [7]:

```
'''
Create the dataset wih digits 1 and 5
'''

def indexDigits(y):
    '''
    Input:
        y: N x 1 2 dimensional numpy array; labels for
handwritten digits
    Output:
        digit_idxs: a dictionary; the keys are digits 0
-- 9
                    for a digit k, digit_idxs[k] is a
```

```
list identifying the rows labeled with digit k
    '''
    N = y.shape[0]
    digit_idxs = {}
    for n in range(N):
        digit = ydigitstrain[n, 0]
        if not digit in digit_idxs:
            digit_idxs[digit] = []
        digit_idxs[digit].append(n)
    return digit_idxs
```

In [8]:

```
'''
Construct the training and test sets for the rest of
the exercises on classifying 1s vs 5s
'''

digit_idxs_train = indexDigits(Xaugtrain)
X1train = Xaugtrain[digit_idxs_train[1], :]
N1train = X1train.shape[0]
print(f'number of 1s: {N1train}')
X5train = Xaugtrain[digit_idxs_train[5], :]
N5train = X5train.shape[0]
print(f'number of 5s: {N5train}')
Xtrain =
Xaugtrain[digit_idxs_train[1]+digit_idxs_train[5], :]
ytrain = np.concatenate((np.ones((N1train, 1)),
-1*np.ones((N5train, 1))), axis=0)
Ntrain, d = Xtrain.shape
print(f'number of 1s and 5s: {Ntrain}')
print(f'Xtrain shape: {Xtrain.shape}, ytrain shape:
{ytrain.shape}')


digit_idxs_test = indexDigits(Xaugtest)
Xtest =
Xaugtest[digit_idxs_test[1]+digit_idxs_test[5], :]
ytest =
np.concatenate((np.ones((len(digit_idxs_test[1]), 1)),
-1*np.ones((len(digit_idxs_test[5]), 1))), axis=0)
Ntest, d = Xtest.shape
```

```python
print(f'number of 1s and 5s: {Ntest}')
print(f'Xtest shape: {Xtest.shape}, ytest shape:
{ytest.shape}')
```

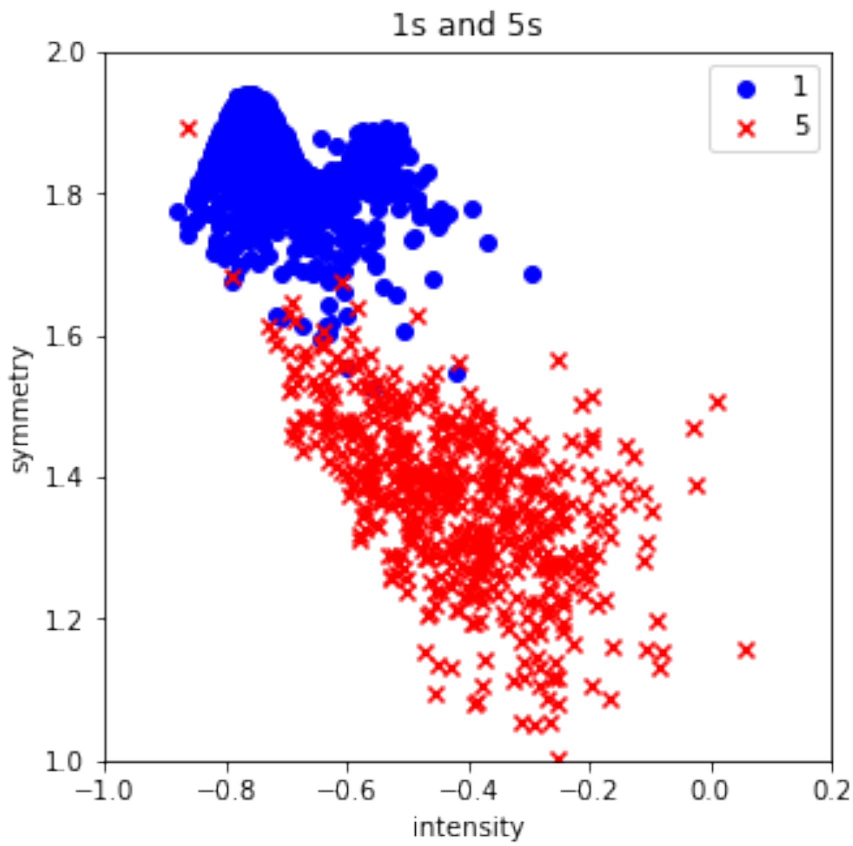
```
number of 1s: 1005
number of 5s: 556
number of 1s and 5s: 1561
Xtrain shape: (1561, 3), ytrain shape: (1561, 1)
number of 1s and 5s: 428
Xtest shape: (428, 3), ytest shape: (428, 1)
```

In [9]:

```python
'''
Plot the training data
'''

fig, axs = plt.subplots(figsize=(5,5))
axs.scatter(X1train[:, 1], X1train[:, 2], marker='o',
color='blue', label='1')
axs.scatter(X5train[:, 1], X5train[:, 2], marker='x',
color='red', label = '5')
axs.set_xlabel('intensity')
axs.set_ylabel('symmetry')
axs.set_xlim(-1, 0.2)
axs.set_ylim(1, 2)
axs.set_title('1s and 5s')
axs.legend()
```

Out[9]:

```
<matplotlib.legend.Legend at 0x21998a93af0>
```



In [10]:

```
'''
Functions to compute the misclassification error
'''

def error(w, x, y, R=None):
    '''
    point-wise error measure for classification
    Input:
        w: a d x 1 2 dimensional numpy array
        x: a d x 1 2 dimensional numpy array
        y: a scalar value
        R: Risk weights; a dictionary
           whose keys are tuples (y, yhat) with
           value equal to the cost of predicting yhat
when the label is y
    Output:
        error: misclassification error of hypothesis w
on data point x with true label y
    '''
```

```
    '''
    TODO: compute the error made by hypothesis with
weights w on data point x with label y
    '''
    return error


def E(w, X, y, R=None):
    '''
    point-wise error measure for classification
    Input:
        w: a d x 1 2 dimensional numpy array
        X: an N x d 2 dimensional numpy array
        y: an N x 1 2 dimensional numpy array
        R: Risk weights; a dictionary
           whose keys are tuples (y, yhat) with
           value equal to the cost of predicting yhat
when the label is y
    Output:
        error: an N x 1 2 dimensional numpy array
               misclassification errors of hypothesis w
on data points in X with true labels y
    '''
    # print(f'w shape {w.shape}, X shape {X.shape}, y
shape {y.shape}')
    N = X.shape[0]
    '''
    TODO: compute the errors made by hypothesis with
weights w on data points in X with true labels y
    '''
    return error
```

In [17]:

```
'''
Helper function to plot a linear separator
'''

def plotLinearSeparator(w, X, y, title=''):
    '''
    Plot data points a linear separator
```

```
    Input:
        w: a d x 1 2 dimensional numpy array
        X: an N x d 2 dimensional numpy array
        y: an N x 1 2 dimensional numpy array
        title: a string
    Output:
        error: misclassification error of hypothesis w
on data points in X with true labels y
    '''
    '''
    Plot data points in X, y
    '''
    plus1s = np.where(y == 1)[0]
    minus1s = np.where(y == -1)[0]
    Xplus1s = X[plus1s, :]
    Xminus1s = X[minus1s, :]
    fig, axs = plt.subplots(figsize=(5,5))
    axs.scatter(Xplus1s[:, 1], Xplus1s[:, 2],
marker='o', color='blue', label='1')
    axs.scatter(Xminus1s[:, 1], Xminus1s[:, 2],
marker='x', color='red', label = '5')
    axs.set_xlabel('intensity')
    axs.set_ylabel('symmetry')
    axs.set_xlim(-1, 0.2)
    axs.set_ylim(1, 2)
    '''
    Plot separator
    '''
    pltxs = np.linspace(-1, 0.21)
    pltys = - (w[0] + w[1] * pltxs) / w[2]
    axs.plot(pltxs, pltys, color='green',
label='separator')
    axs.set_title(title)
    axs.legend()
    plt.show()
```

In [18]:

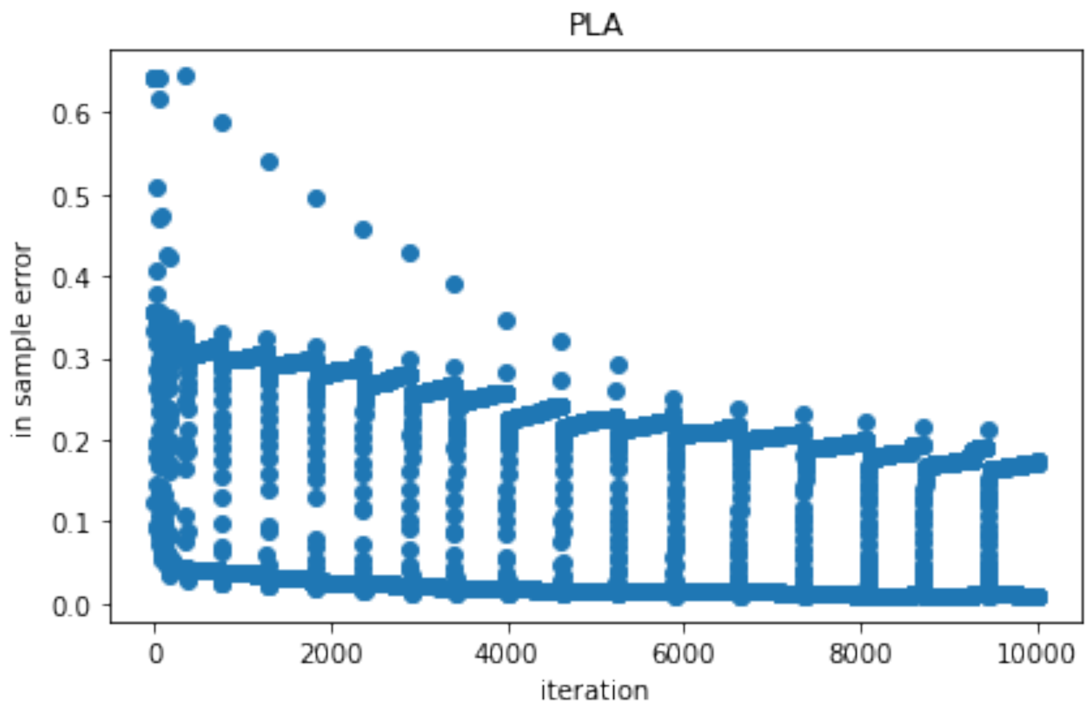
```
'''
The Pocket algorithm (variant of the Perceptron
Learning Algorithm)
'''
```

```
def pocket(X, y, max_iters=1000, w_init=None):
    '''
    Implements the Pocket algorithm
    Input:
        X: A 2 dimensional N x d numpy array
           The i-th row X[i, :] contains features for
the i-th example in the training set
           X[i, 0] = 1
           X[i, 1], ... X[i, d] have values of features
        y: A 2 dimensional N x d numpy array
           y[i, 0] is the label associated with the i-
th example
        max_iters: an integer; maximum number of
iterations of PLA
        w_init: A 2 dimensional d x 1 numpy array
                intended to set initial weights for PLA
    Output:
        w_best: a d x 1 2 dimensional numpy array
                weights with lowest error on the input
training set X, y
    '''
    Eins = []
    ws = []

    Ein_best = np.infty
    w_best = 0

    w = np.zeros((d, 1))
    w = w + 0.0000001
    if not isinstance(w_init, type(None)):
        w = w_init

    for i in range(max_iters):
        Ein = E(w, X, y)
        Eins.append(Ein)
        ws.append(w)
        if Ein < Ein_best:
            Ein_best = Ein
            w_best = w
```

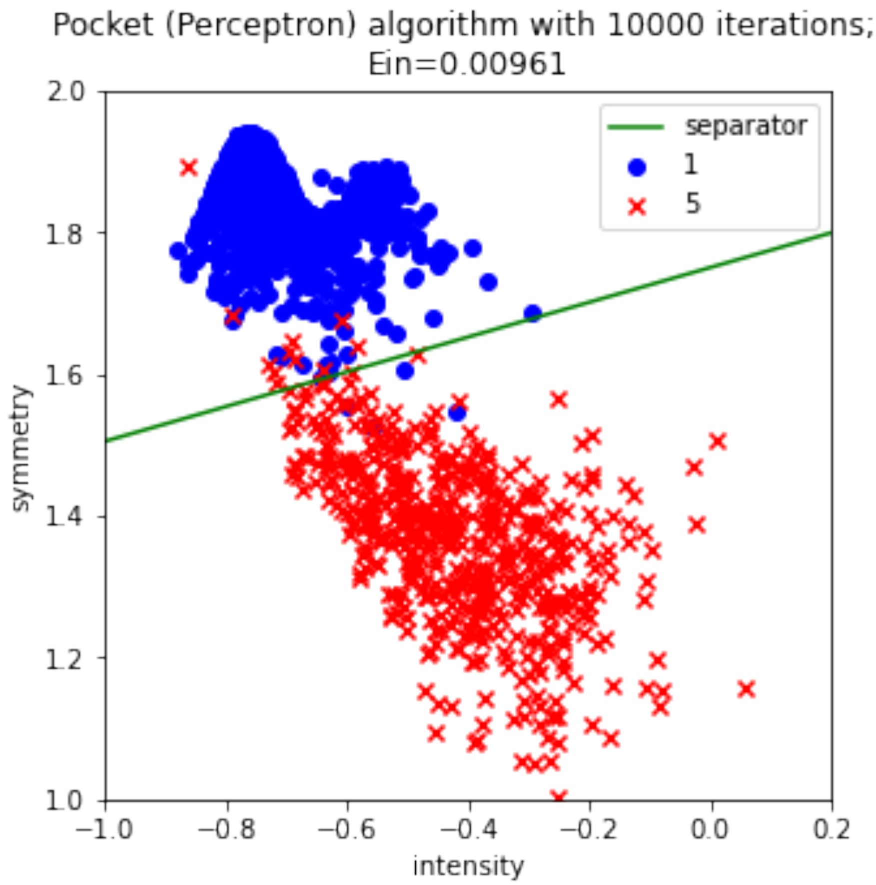
```
        '''
        TODO: Complete this implementation of the
Pocket algorithm
        '''

    plt.scatter(range(max_iters), Eins)
    plt.xlabel('iteration')
    plt.ylabel('in sample error')
    plt.title('PLA')
    plt.tight_layout()
    plt.show()

    print(f'Ein_best {Ein_best}, \nw_best \n{w_best}')

    return w_best
```

In [20]:

```
'''
Run the Pocket algorithm
'''

max_iters = 10000
w = pocket(Xtrain, ytrain, max_iters=max_iters)
Ein = np.round(E(w, Xtrain, ytrain),5)
plotLinearSeparator(w, Xtrain, ytrain, title=f'Pocket
(Perceptron) algorithm with {max_iters} iterations;
\nEin={Ein}')
```




```
Ein_best 0.009609224855861626,
w_best
[[-20.9999999 ]
 [ -2.9436249 ]
 [ 11.99648448]]
```



In [24]:

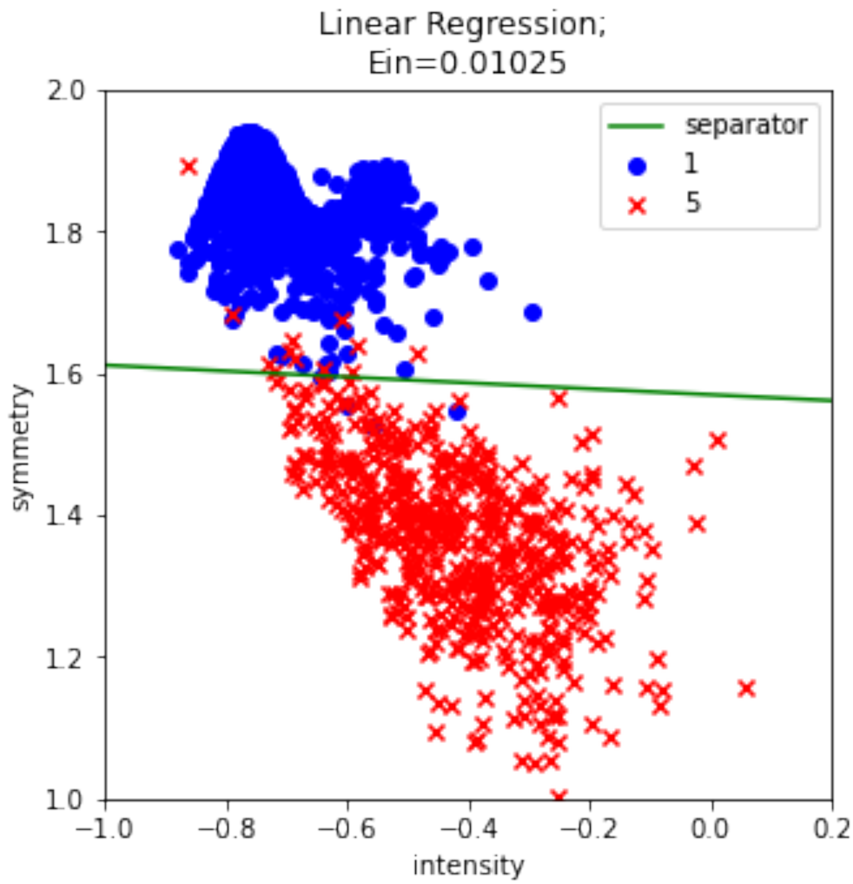
```
'''
The one-step optimal algorithm for Linear Regression
(See LFD Section 3.2.1)
'''
def linearRegression(X, y):
    '''
    Implements the one-step algorithm for Linear
Regression (See LFD Section 3.2.1)
    Input:
        X: A 2 dimensional N x d numpy array
           The i-th row X[i, :] contains features for
the i-th example in the training set
           X[i, 0] = 1
           X[i, 1], ... X[i, d] have values of features
        y: A 2 dimensional N x d numpy array
           y[i, 0] is the label associated with the i-
th example
    Output:
```

```
        w: a d x 1 2 dimensional numpy array
           weights with lowest error on the input
training set X, y
    '''
    '''
    TODO: Implement the one-step optimal algorithm for
linear regression
    '''
    print(f'Ein {Ein}, \nw_lin \n{w_lin}')
    return w_lin
```

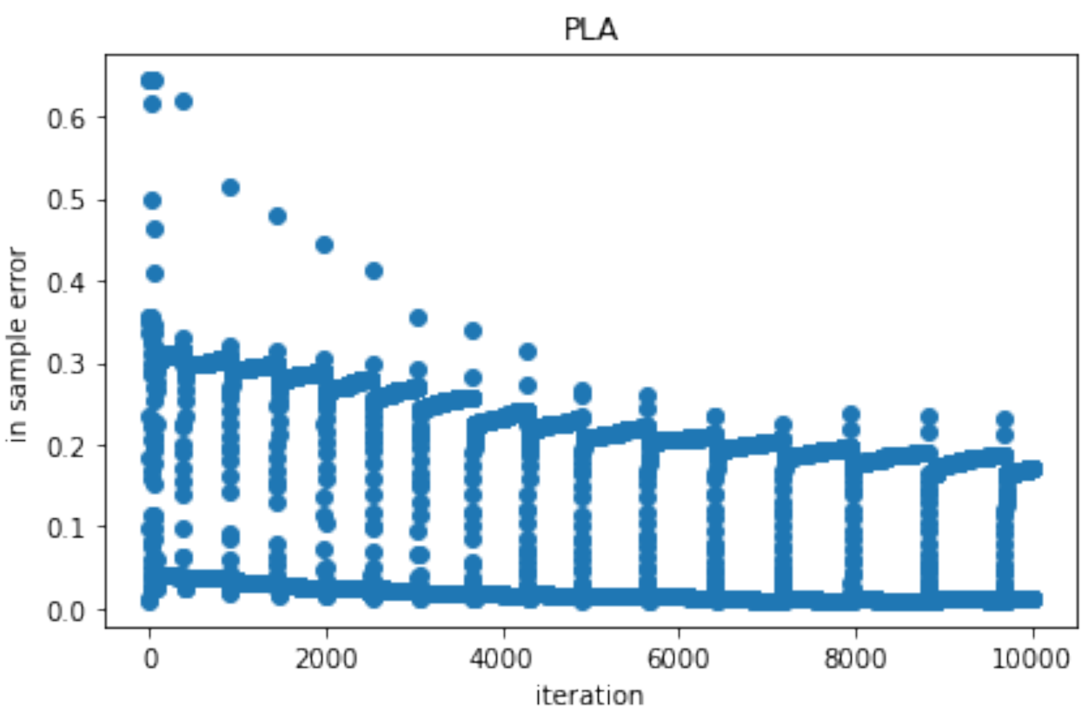
In [25]:

```
'''
Run Linear Regression followed by the Pocket algorithm
to classify 1s vs 5s
'''
w_lin = linearRegression(Xtrain, ytrain)
Ein = np.round(E(w_lin, Xtrain, ytrain), 5)
plotLinearSeparator(w_lin, Xtrain, ytrain,
title=f'Linear Regression; \nEin={Ein}')

max_iters = 10000
w = pocket(Xtrain, ytrain, w_init=w_lin,
max_iters=max_iters)
Ein = np.round(E(w, Xtrain, ytrain), 5)
plotLinearSeparator(w, Xtrain, ytrain, \
                    title=f'Pocket algorithm after
Linear Regression with {max_iters} iterations;
\nEin={Ein}')
```

```
Ein 0.010249839846252402,
w_lin
[[-6.05415811]
 [ 0.16071908]
 [ 3.85642483]]
```

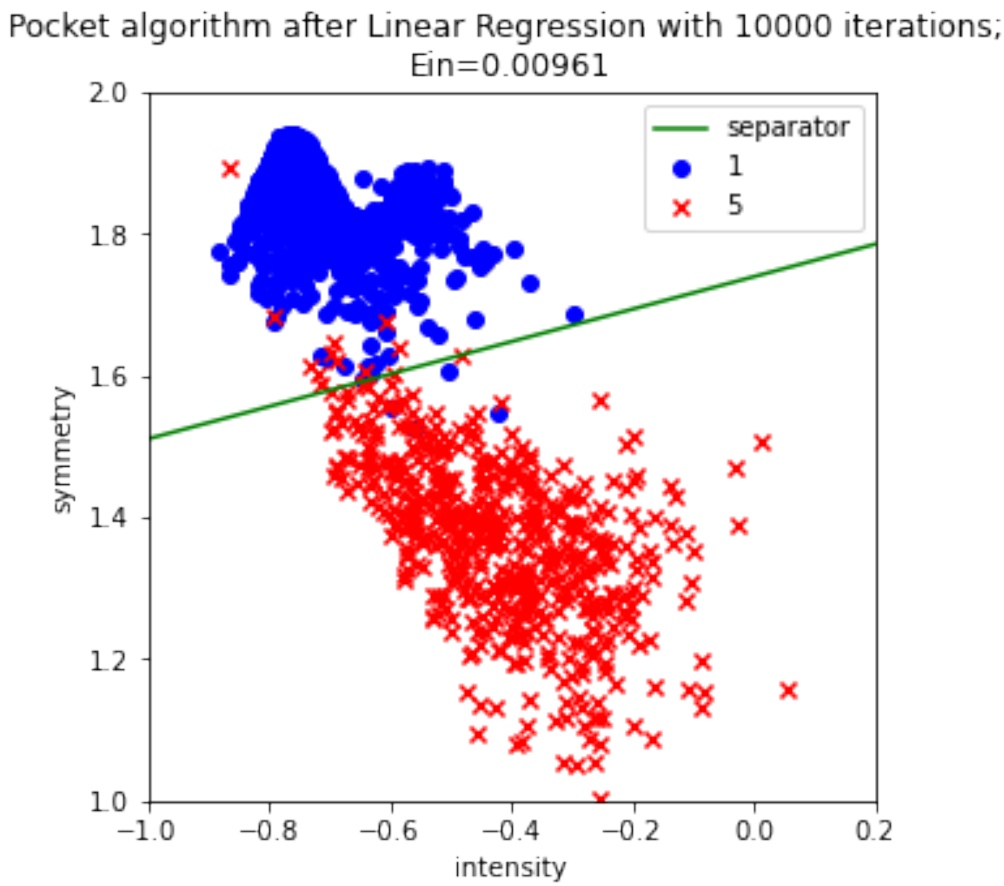


Linear Regression;
Ein=0.01025
separator
1
5
symmetry
intensity
2.0
1.8
1.6
1.4
1.2
1.0
−1.0
−0.8
−0.6
−0.4
−0.2
0.0
0.2



PLA
in sample error
iteration
0.6
0.5
0.4
0.3
0.2
0.1
0.0
0
2000
4000
6000
8000
10000

```
Ein_best 0.009609224855861626,
w_best
[[-21.05415811]
 [ -2.77235905]
 [ 12.10354983]]
```

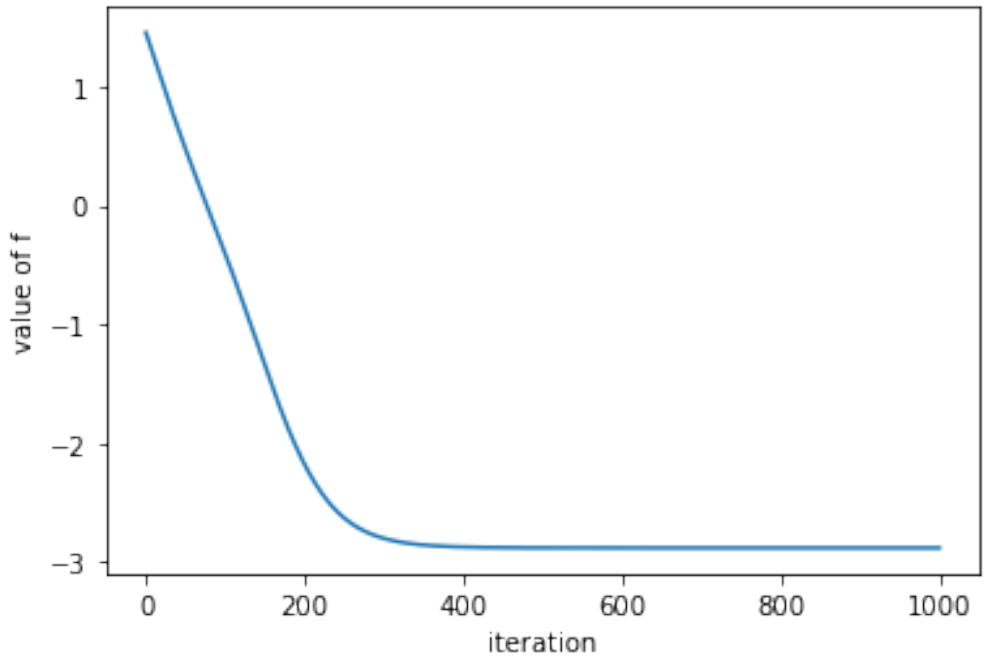


In [26]:

```python
'''
Gradient descent to minimize an arbitrary function
'''

def functionf(x, y):
    '''
    Computes the value of an arbitrary function in two variables at the input location
    '''
    '''
```

```
    TODO: Compute the value of the function at point x,
y
    '''
    return fval


def gradientf(x, y):
    '''
    Computes the gradient of an arbitrary function in
two variables at the input location
    '''
    '''
    TODO: Compute the gradient
    '''
    return df_by_dx, df_by_dy


def gradientDescent4f(x, y, eta=0.001, max_iters=100):
    '''
    Performs gradient descent to find the location at
which the value of an arbitrary function is minimized
    '''
    fvals = []
    for i in range(max_iters):
        # print(f'iteration {i}, x={x}, y={y}')
        fval = functionf(x, y)
        fvals.append(fval)
        grad = gradientf(x, y)
        '''
        TODO: Complete this implementation of gradient
descent for an arbitrary function f with two variables
        '''
    plt.plot(range(max_iters), fvals)
    plt.xlabel('iteration')
    plt.ylabel('value of f')
    plt.show()
    return x, y

max_iters = 1000
eta = 0.0001
```

```
gradientDescent4f(0.1, 0.1, eta=eta,
max_iters=max_iters)
```



Out[26]:

```
(-0.24182360040042794, 4.698413037027736e-06)
```

In [27]:

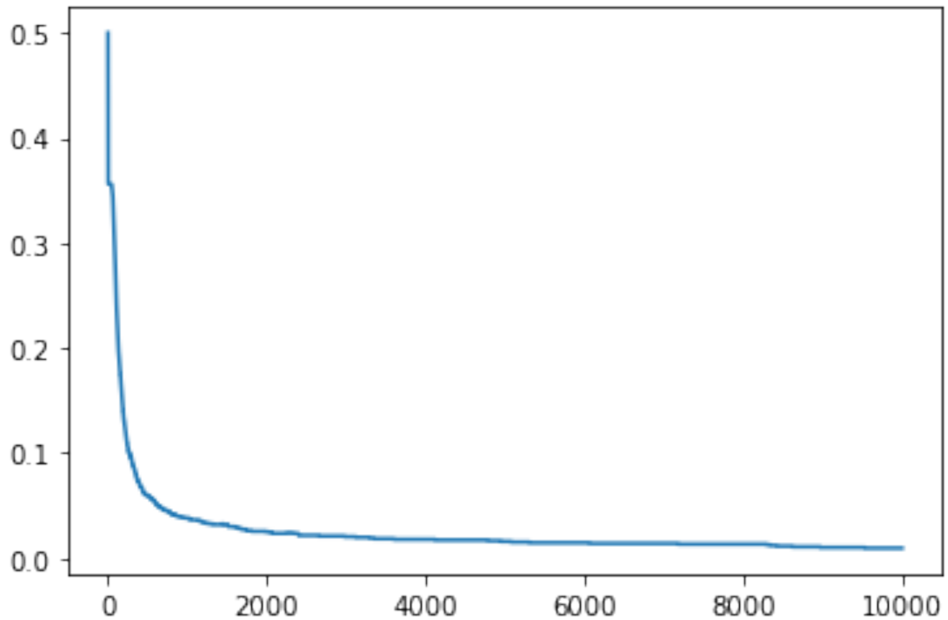
```
'''
The Logistic Regression algorithm with the cross
entropy error measure
'''

def sigmoid(s):
    '''
    Implements the sigmoid function
    '''
    return 1 / (1 + np.exp(-1*s))


def gradientCrossEntropyError(w, X, y):
    '''
    Computes the gradient of the cross entropy error
function on the dataset X, y at input weights w
    See LFD Exercise 3.7
    Input:
```

```
        w: a d x 1 2 dimensional numpy array
        X: an N x d 2 dimensional numpy array
        y: an N x 1 2 dimensional numpy array
    Output:
        gradient: a d x 1 2 dimensional numpy array
                  gradient of the cross entropy error
function on the dataset X, y at input weights w
    '''
    N, d = X.shape
    grad = np.zeros((d, 1))
    for n in range(N):
        y_n = y[n, 0]
        x_n = X[n, :].reshape((d, 1))
        '''
        TODO: Complete this implementation to compute
the gradient at input weights w
        '''
    return grad


def logisticRegression(X, y, eta=0.001, w_init=None,
max_iters=1000):
    '''
    Implements the gradient descent algorithm for
Logistic Regression
    See LFD Example 3.3
    Input:
        X: A 2 dimensional N x d numpy array
           The i-th row X[i, :] contains features for
the i-th example in the training set
           X[i, 0] = 1
           X[i, 1], ... X[i, d] have values of features
        y: A 2 dimensional N x d numpy array
           y[i, 0] is the label associated with the i-
th example
        eta: learning rate
        w_init: a d x 1 2 dimensional numpy array
                initial weights to start gradient
descent
        max_iters: maxmimum number of iterations of
```

```
gradient descent
    Output:
        w: a d x 1 2 dimensional numpy array
           weights with (approximately) lowest error on
the input training set X, y
    '''
    N, d = X.shape
    w = np.zeros((d, 1))
    if not isinstance(w_init, type(None)):
        w = w_init
    Eins = []
    for i in range(max_iters):
        Ein = E(w, X, y)
        Eins.append(Ein)
        grad = gradientCrossEntropyError(w, X, y)
        '''
        TODO: Complete this implementation of the
gradient descent algorithm for logistic regression
        '''
    plt.plot(range(max_iters), Eins)
    return w
```

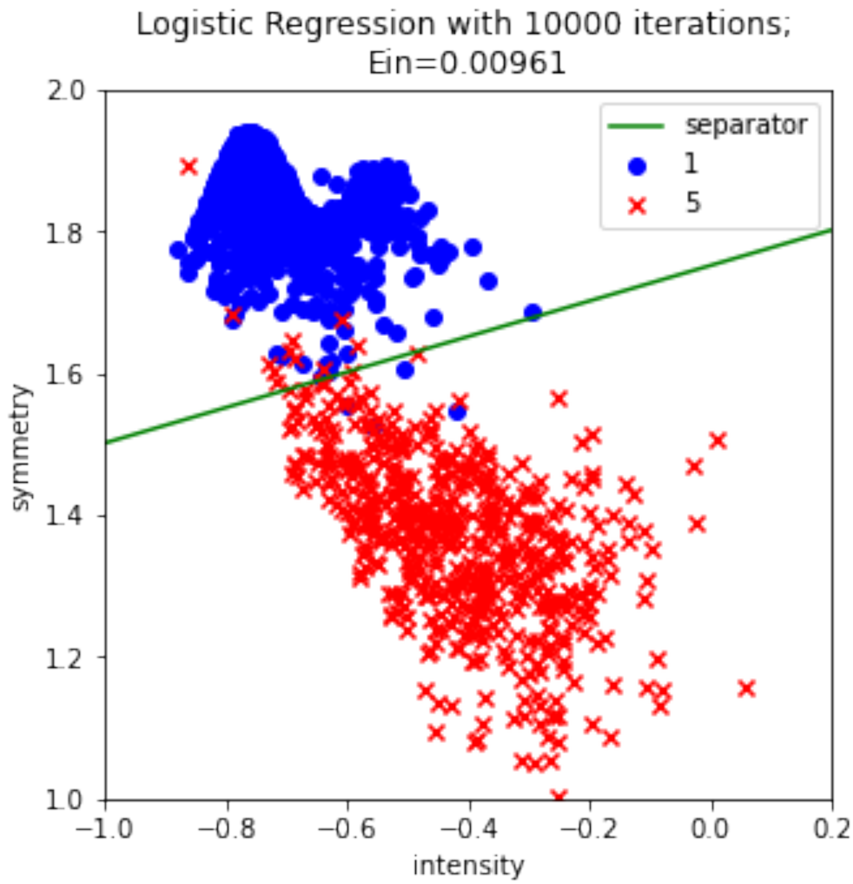
In [28]:

```
'''
Run the logistic regression algorithm to classify 1s vs
5s
'''

max_iters = 10000
eta = 0.5
w = logisticRegression(Xtrain, ytrain, eta=eta,
max_iters=max_iters)
```



In [29]:

```
plotLinearSeparator(w, Xtrain, ytrain, title=f'Logistic
Regression with {max_iters} iterations; \nEin={Ein}')
```


Logistic Regression with 10000 iterations;
Ein=0.00961
separator
1
5
symmetry
intensity
2.0
1.8
1.6
1.4
1.2
1.0
−1.0
−0.8
−0.6
−0.4
−0.2
0.0
0.2