ep03_logistic_regression

June 12, 2021

[1]:
```
name = "Renata Sarmet Smiderle Mendes"  # write YOUR NAME

honorPledge = "I affirm that I have not given or received any unauthorized " \
              "help on this assignment, and that this work is my own.\n"


print("\nName: ", name)
print("\nHonor pledge: ", honorPledge)
```

```
Name:  Renata Sarmet Smiderle Mendes

Honor pledge:  I affirm that I have not given or received any unauthorized help
on this assignment, and that this work is my own.
```

1 MAC0460 / MAC5832 (2021)

1.1 EP3: Logistic regression

1.1.1 Topics / concepts explored in this EP:

- Implementation of the **logistic regression algorithm**, using the gradient descent technique
- Application on binary classification of 2D examples (i.e., $d = 2$)
- Confusion matrix, effects of unbalanced classes

Complete and submit this notebook. Places to be filled are indicated with this color

1.1.2 Evaluation criteria

- Correctitude of the algorithms
- Code
  - do not change the prototype of the functions
  - efficiency (you should avoid unnecessary loops; use matrix/vector computation with NumPy wherever appropriate)
  - cleanliness (do not leave any commented code or useless variables)
- Appropriateness of the answers
- File format: Complete and submit this notebook with the outputs of the execution. **Do no change the file name.**

1.1.3 Hints

- In this notebook vectors are implemented as ndarray (n,) and not as ndarray (n,1)
- It might be wise to first make sure your implementation is correct. For instance, you can compare the results of your implementation with the ones obtained with the implementation available in scikit-learn. After you feel confident, paste your code in the notebook, and then run the rest of the code in the notebook.
- If you face difficulties with Keras or other libraries used in this notebook, as well as clarity issues in the exercises in this notebook, post a message in the Forum for discussions.

2 The algorithm

We will use the formulation described in the textbook (Learning from data, Abu-Mostafa et al.). Positive class label is equal to 1 and negative class label is equal to -1.

The loss (or cost) function to be minimized is

$$E_{in}(\mathbf{w}) = \frac{1}{N}\sum_{i=1}^{N} \ln(1 + e^{-y^{(i)}\mathbf{w}^T\mathbf{x}^{(i)}}) \tag{1}$$

Its gradient is given by

$$\nabla E_{in}(\mathbf{w}) = -\frac{1}{N}\sum_{i=1}^{N} \frac{y^{(i)}\mathbf{x}^{(i)}}{1 + e^{y^{(i)}\mathbf{w}^T\mathbf{x}^{(i)}}} \tag{2}$$

The logistic (sigmoid) function is

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{3}$$

[2]:
```
# All imports

import numpy as np
import pandas as pd
import seaborn as sb

import matplotlib.pyplot as plt
from matplotlib.lines import Line2D

from sklearn.datasets import make_blobs
from sklearn.preprocessing import LabelEncoder

from tensorflow.keras.datasets import mnist

%matplotlib inline
```

3 1. Training and prediction algorithms

In the next four code cells, write the code to implement the specified functions. These functions will be used below for logistic regression training and prediction, in Sections 2 to 4. Use vectorial computation with NumPy.

3.1 1.1. Cross-entropy loss

This is Equation (1) above.

[3]:
```
def cross_entropy_loss(w, X, y):
    """
    Computes the loss (equation 1)
    :param w: weight vector
    :type: np.ndarray(shape=(1+d, ))
    :param X: design matrix
    :type X: np.ndarray(shape=(N, 1+d))
    :param y: class labels
    :type y: np.ndarray(shape=(N, ))
    :return loss: loss (equation 1)
    :rtype: float
    """

    loss = np.log(np.exp(np.dot(w,X.T)*-y)+1).mean()
    return loss
```

3.2 1.2. Gradient of the cross-entropy loss

This is Equation (2) above.

[4]:
```
def cross_entropy_gradient(w, X, y):
    """
    Computes the gradient of the loss function (equation 2)
    :param w: weight vector
    :type: np.ndarray(shape=(1+d, ))
    :param X: design matrix
    :type X: np.ndarray(shape=(N, 1+d))
    :param y: class labels
    :type y: np.ndarray(shape=(N, ))
    :return grad: gradient (equation 2)
    :rtype: np.ndarray(shape=(1+d, ))
    """

    gradient = -np.dot((y/(np.exp(np.dot(w,X.T)*y)+1)),X)/X.shape[0]
    return gradient
```

3.3 1.3 Logistic regression training

The function below receives the data matrix X (shape = (N, d)) and the ouput vector y (shape = (N,)), and should return the final weight vector w (shape = (d+1,)) and, optionally (when parameter return_history = True), a list of size num_iterations+1 with the cross-entropy loss values at the beginning and after each of the iterations.

Note that the data matrix needs to be extended with a column of 1's.

If w0==None it must be initialized with w0 = np.random.normal(loc = 0, scale = 1, size = X.shape[1])

[5]:
```
def train_logistic(X, y, learning_rate = 1e-1, w0 = None,\
                        num_iterations = 1000, return_history = False):
    """
    Computes the weight vector applying the gradient descent technique
    :param X: design matrix
    :type X: np.ndarray(shape=(N, d))
    :param y: class label
    :type y: np.ndarray(shape=(N, ))
    :return: weight vector
    :rtype: np.ndarray(shape=(1+d, ))
    :return: the history of loss values (optional)
    :rtype: list of float
    """


    # add a left column with 1's into X -- X extended,
    # that way Xi has the same number of elements of the weight array
    Xe = np.hstack((np.ones((X.shape[0],1)), X))

    # Initialize w0 ramdomly if it is None
    if w0 is None:
         w0 = np.random.normal(loc = 0, scale = 1, size = Xe.shape[1])

    # Initialize w
    w = w0

    # Initialize loss values list
    list_loss = []

    for t in range(num_iterations):
        # Compute the gradient
        g = cross_entropy_gradient(w, Xe, y)

        # Set the direction to move
        v = -g

        # Update the weights
```

```
        w = w + learning_rate*v

        if return_history:
            list_loss.append(cross_entropy_loss(w, Xe, y))

    return w, list_loss
```

3.4 1.4. Logistic regression prediction

The function in the next cell will be used to do the prediction of logistic regression. Recall that the prediction is a score in $[0, 1]$, given by the sigmoid value of the linear combination.

[6]:
```
def sigmoid(z):
    return 1 / (1 + np.exp(-z))


def predict_logistic(X, w):
    """
    Computes the logistic regression prediction
    :param X: design matrix
    :type X: np.ndarray(shape=(N,d))
    :param w: weight vector
    :rtype: np.ndarray(shape=(1+d,))
    :return: predicted classes
    :rtype: np.ndarray(shape=(N,))
    """


    # add a left column with 1's into X -- X extended,
    # that way Xi has the same number of elements of the weight array
    Xe = np.hstack((np.ones((X.shape[0],1)), X))

    # predict
    y = sigmoid(np.dot(Xe,w))

    return y
```

4 2. Testing on a toy dataset

4.1 2.1. Generate two blobs of points

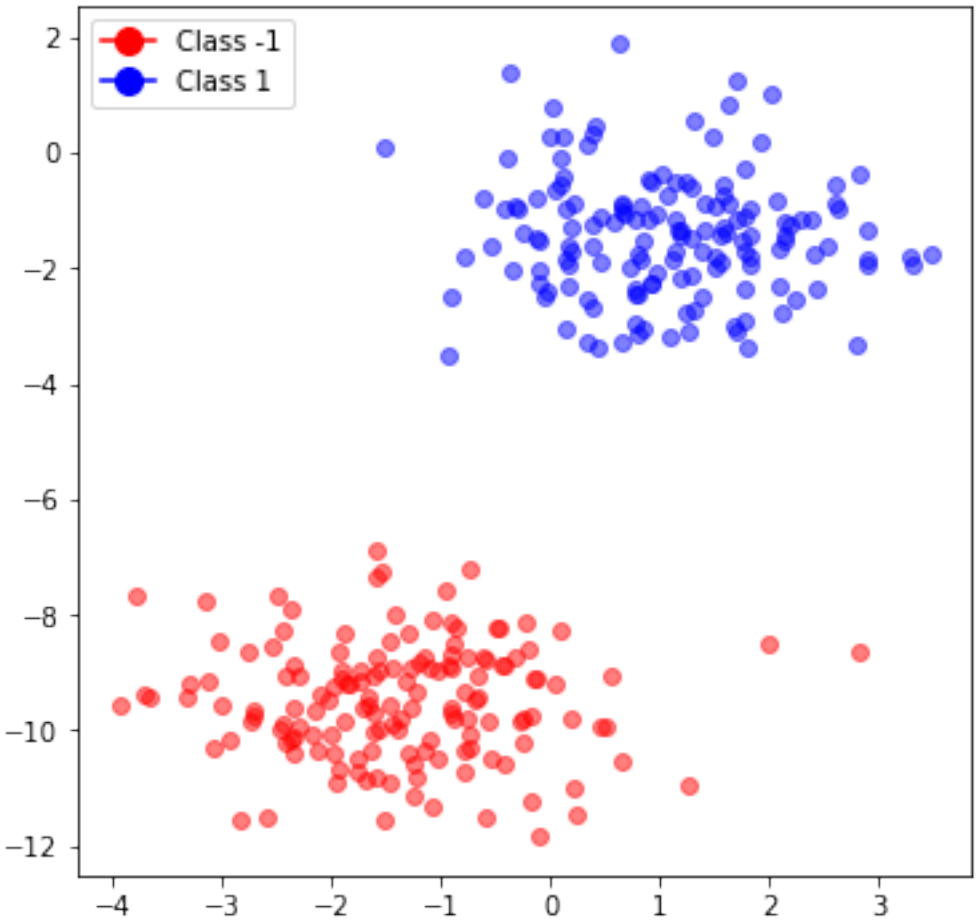
[7]:
```
# Create two blobs
N = 300
X, y = make_blobs(n_samples=N, centers=2, cluster_std=1, n_features=2,␣
 ↪random_state=2)
```

```
# change labels 0 to -1
y[y==0] = -1

print("X.shape =", X.shape, "  y.shape =", y.shape)
```

```
X.shape = (300, 2)   y.shape = (300,)
```

4.2 2.2. Let's plot the blobs of points

[8]:
```
fig = plt.figure(figsize=(6,6))

# plot negatives in red
plt.scatter(X[y==-1,0], \
            X[y==-1,1], \
            alpha = 0.5,\
            c = 'red')

# and positives in blue
plt.scatter(x=X[y==1,0], \
            y=X[y==1,1], \
            alpha = 0.5, \
            c = 'blue')

P=+1
N=-1
legend_elements = [ Line2D([0], [0], marker='o', color='r',\
                    label='Class %d'%N, markerfacecolor='r',\
                    markersize=10),\
                    Line2D([0], [0], marker='o', color='b',\
                    label='Class %d'%P, markerfacecolor='b',\
                    markersize=10) ]

plt.legend(handles=legend_elements, loc='best')
plt.show()
```

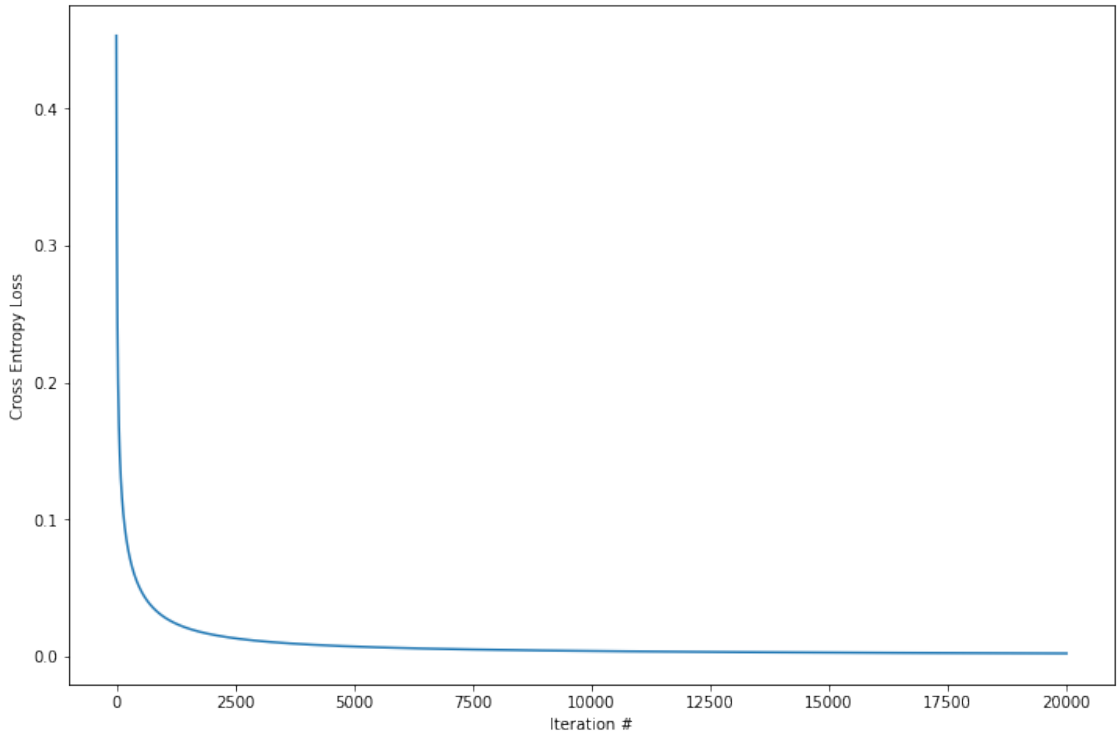


4.3 2.3. Let's train the linear regressor and plot the loss curve

[9]:
```
np.random.seed(567)

# ==> Replace the right hand side below with a call to the
# train_logistic() function defined above. Use parameter return_history=True

w_logistic, loss = train_logistic(X, y, num_iterations = 20000, return_history␣
↪= True)

# ==> Your code insert ends here

print()
print("Final weight:\n", w_logistic)
print()
print("Final loss:\n", loss[-1])
```

```
plt.figure(figsize = (12, 8))
plt.plot(loss)
plt.xlabel('Iteration #')
plt.ylabel('Cross Entropy Loss')
plt.show()
```

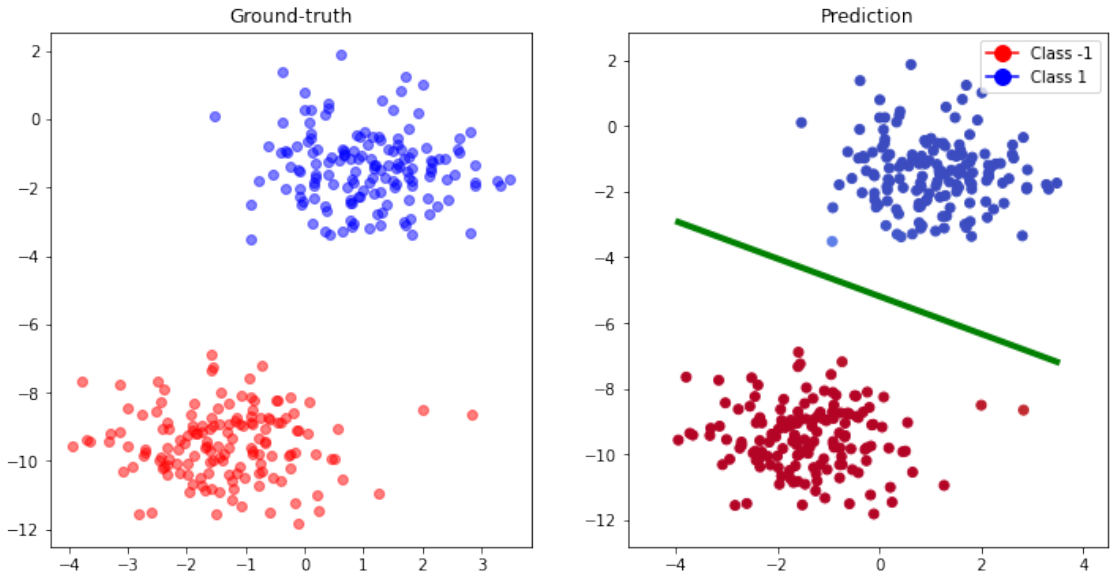
```
Final weight:
 [9.09104395 1.00519642 1.75122646]

Final loss:
 0.002052776301488329
```



4.4 2.4. Now, let's plot the decision boundary

[10]:
```
x1min = min(X[:,0])
x1max = max(X[:,0])
x2min = min(X[:,1])
x2max = max(X[:,1])

y_pred = predict_logistic(X, w_logistic)

fig = plt.figure(figsize=(12,6))
```

```
ax1 = fig.add_subplot(121)
ax1.set_title("Ground-truth")

# plot negatives in red
ax1.scatter(X[y==-1,0], \
            X[y==-1,1], \
            alpha = 0.5, \
            c = 'red')

# and positives in blue
ax1.scatter(x=X[y==1,0], \
            y=X[y==1,1], \
            alpha = 0.5, \
            c = 'blue')

ax2 = fig.add_subplot(122)

ax2.set_title("Prediction")
ax2.scatter(x = X[:,0], y = X[:,1], c = -y_pred, cmap = 'coolwarm')
ax2.legend(handles=legend_elements, loc='best')
ax2.set_xlim([x1min-1, x1max+1])
ax2.set_ylim([x2min-1, x2max+1])

p1 = (x1min, -(w_logistic[0] + (x1min)*w_logistic[1])/w_logistic[2])
p2 = (x1max, -(w_logistic[0] + (x1max)*w_logistic[1])/w_logistic[2])

lines = ax2.plot([p1[0], p2[0]], [p1[1], p2[1]], '-')
plt.setp(lines, color='g', linewidth=4.0)

plt.show()
```



5 3. Testing on the data we have collected

5.1 3.1. Read and prepare the dataset

[11]:
```
# load the dataset
df = pd.read_csv('QT1data.csv')
df.head()
```

[11]:

	Sex	Age	Height	Weight	Shoe number	Trouser number
0	Female	53	154	59	36	40
1	Male	23	170	56	40	38
2	Female	23	167	63	37	40
3	Male	21	178	78	40	40
4	Female	25	153	58	36	38

[12]:
```
df.describe()
```

[12]:

	Age	Height	Weight	Shoe number
count	130.000000	130.000000	130.000000	130.000000
mean	28.238462	170.684615	70.238462	39.507692
std	12.387042	11.568491	15.534809	2.973386
min	3.000000	100.000000	15.000000	24.000000
25%	21.000000	164.250000	60.000000	38.000000
50%	23.000000	172.000000	69.500000	40.000000
75%	29.000000	178.000000	80.000000	41.000000
max	62.000000	194.000000	130.000000	46.000000

[13]:
```
# Does filtering out the 'children' make any difference ? Not that much
# df = df[df['Height']>130]
```

[14]:
```
# Select only features of interest
feature_cols = ['Height', 'Weight']
X = (df.loc[:, feature_cols]).to_numpy(dtype=float)

# Input normalization
for i in range(X.shape[1]):
    avg = np.mean(X[:, i])
    stddev = np.std(X[:, i])
    X[:, i] = (X[:, i] - avg) / stddev

print(X.shape)
```

```
(130, 2)
```

[15]:
```
# Our target variable is Sex
sex = df.pop('Sex').values
```

[16]:
```
# convert to negative=-1 and positive=1 using functions from scikit-learn
label_encoder = LabelEncoder()
y = label_encoder.fit_transform(sex)
y = np.where(y==0, -1, y)
print(y.shape)
print(min(y), max(y))
```

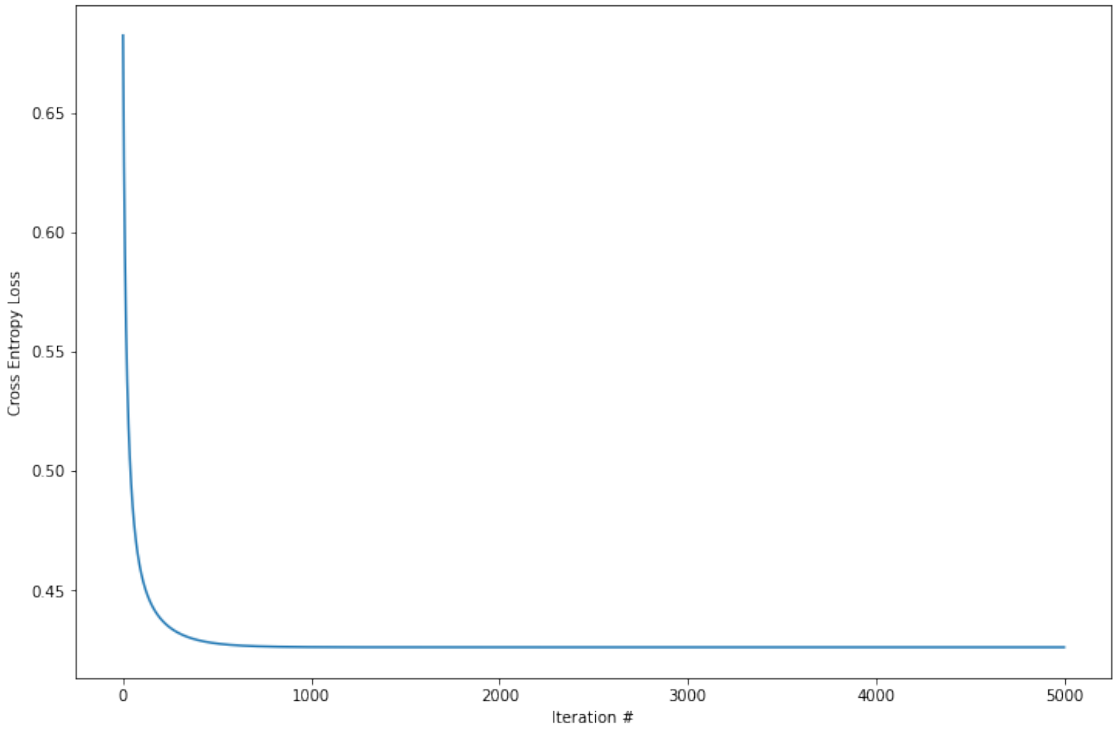
```
(130,)
-1 1
```

5.2 3.2. Training

[17]:
```
# ===> Write the code for
#      - training,
#      - printing the final weight vector
#      - plotting the loss curve
np.random.seed(567)

w_logistic, loss = train_logistic(X, y, num_iterations = 5000, return_history =␣
 ↪True)

# ==> Your code insert ends here

print()
print("Final weight:\n", w_logistic)
print()
print("Final loss:\n", loss[-1])
```

```
plt.figure(figsize = (12, 8))
plt.plot(loss)
plt.xlabel('Iteration #')
plt.ylabel('Cross Entropy Loss')
plt.show()
```

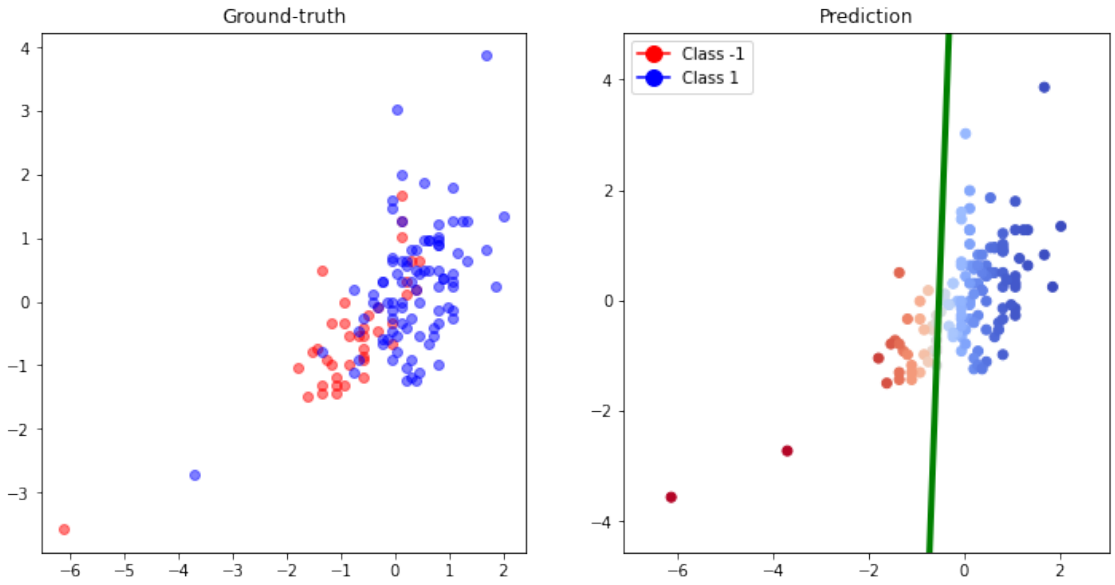
```
Final weight:
 [ 1.13699938  2.14343309 -0.09267794]

Final loss:
 0.42614890001352657
```



5.3 3.3. Ploting the ground-truth, prediction + decision boundary

[18]:
```
# ===> Write the code for
#      - computing the prediction
#      - plotting the scatterplot with the ground-truth
#        and another with the prediction and decision boundary line

x1min = min(X[:,0])
x1max = max(X[:,0])
```

```
x2min = min(X[:,1])
x2max = max(X[:,1])

y_pred = predict_logistic(X, w_logistic)

fig = plt.figure(figsize=(12,6))
ax1 = fig.add_subplot(121)
ax1.set_title("Ground-truth")

# plot negatives in red
ax1.scatter(X[y==-1,0], \
            X[y==-1,1], \
            alpha = 0.5, \
            c = 'red')

# and positives in blue
ax1.scatter(x=X[y==1,0], \
            y=X[y==1,1], \
            alpha = 0.5, \
            c = 'blue')

ax2 = fig.add_subplot(122)

ax2.set_title("Prediction")
ax2.scatter(x = X[:,0], y = X[:,1], c = -y_pred, cmap = 'coolwarm')
ax2.legend(handles=legend_elements, loc='best')
ax2.set_xlim([x1min-1, x1max+1])
ax2.set_ylim([x2min-1, x2max+1])

p1 = (x1min, -(w_logistic[0] + (x1min)*w_logistic[1])/w_logistic[2])
p2 = (x1max, -(w_logistic[0] + (x1max)*w_logistic[1])/w_logistic[2])

lines = ax2.plot([p1[0], p2[0]], [p1[1], p2[1]], '-')
plt.setp(lines, color='g', linewidth=4.0)

plt.show()
```



5.4 3.4 How good is the separation?

Based on the predicted probabilities, we can decide the final class label for each example $\mathbf{x}$ as follows:

$$\text{class label of } \mathbf{x} = \begin{cases} +1, & \text{if } \hat{P}(y=1|\mathbf{x}) > 0.5, \\ -1, & \text{if } \hat{P}(y=1|\mathbf{x}) \leq 0.5 \end{cases}$$

[19]:
```
# ===> write your code to compute how many wrong
#      classifications we would have if we use the decision rule above

def define_class(y, threshold=0.5):

    classifier = lambda t: 1 if t > threshold else -1
    vfunc = np.vectorize(classifier)
    return vfunc(y)


threshold = 0.5

y_class = define_class(y_pred, threshold)

misclassified = np.where(y != y_class)[0]

print(f'We have {misclassified.size} misclassified points using {threshold} as␣
 ↪threshold.')
```

```
We have 21 misclassified points using 0.5 as threshold.
```

5.5 3.4 Repeat for $d > 2$ variables

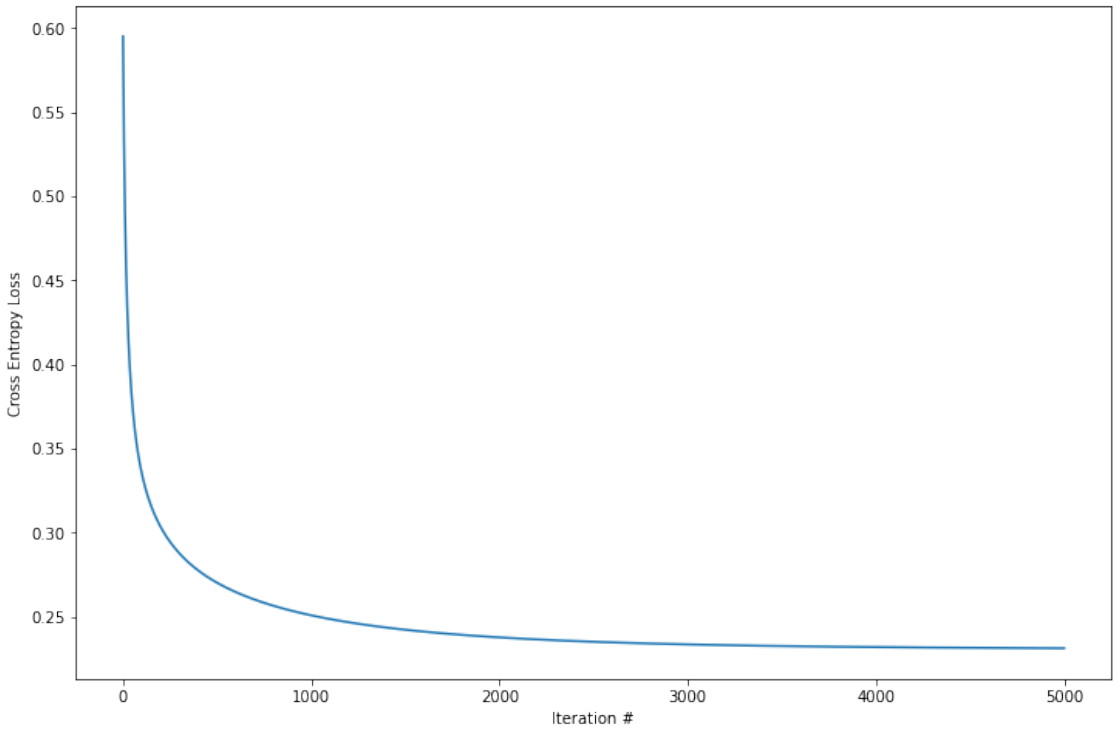
In this case, there is no need to display the scatter plot.

(Just for fun)

[20]:
```python
# ===> Your code

# load the dataset
df = pd.read_csv('QT1data.csv')

# Select only features of interest
feature_cols = ['Height', 'Weight', 'Shoe number']
print(f"Using features = {feature_cols}\n")
X = (df.loc[:, feature_cols]).to_numpy(dtype=float)

# Input normalization
for i in range(X.shape[1]):
    avg = np.mean(X[:, i])
    stddev = np.std(X[:, i])
    X[:, i] = (X[:, i] - avg) / stddev

print(X.shape)

# Our target variable is Sex
sex = df.pop('Sex').values

# convert to negative=-1 and positive=1 using functions from scikit-learn
label_encoder = LabelEncoder()
y = label_encoder.fit_transform(sex)
y = np.where(y==0, -1, y)
print(y.shape)
print(min(y), max(y))

#       - training,
#       - printing the final weight vector
np.random.seed(567)

w_logistic, loss = train_logistic(X, y, num_iterations = 5000, return_history =␣
 ↪True)


print()
print("Final weight:\n", w_logistic)
print()
print("Final loss:\n", loss[-1])
```

```
#       - computing the prediction and getting the misclassified points

y_pred = predict_logistic(X, w_logistic)

threshold = 0.5
y_class = define_class(y_pred, threshold)

misclassified = np.where(y != y_class)[0]

print(f'\nWe have {misclassified.size} misclassified points using {threshold}␣
 ↪as threshold.')

#       - plotting the loss curve

plt.figure(figsize = (12, 8))
plt.plot(loss)
plt.xlabel('Iteration #')
plt.ylabel('Cross Entropy Loss')
plt.show()
```

```
Using features = ['Height', 'Weight', 'Shoe number']

(130, 3)
(130,)
-1 1

Final weight:
 [ 2.17246518 -1.1932779  -0.9610623   5.5698059 ]

Final loss:
 0.23112668538067496

We have 14 misclassified points using 0.5 as threshold.
```

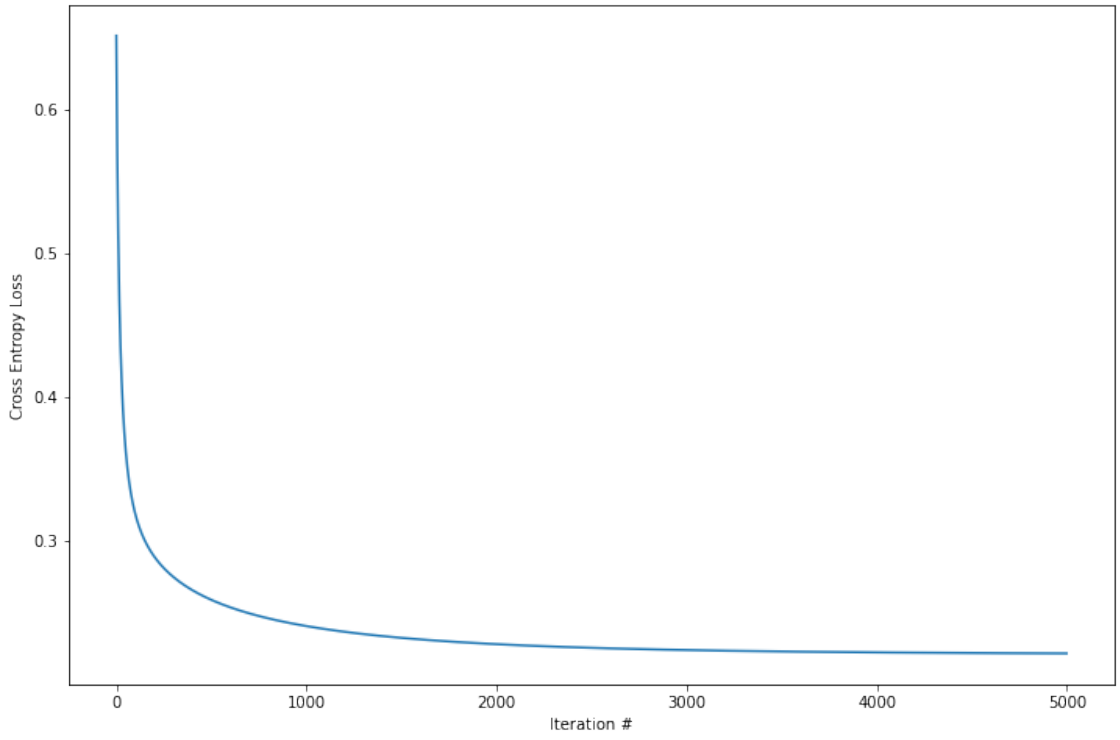


[21]:
```
# ===> Your code

# load the dataset
df = pd.read_csv('QT1data.csv')

# Select only features of interest
feature_cols = ['Height', 'Weight', 'Shoe number', 'Age']
print(f"Using features = {feature_cols}\n")
X = (df.loc[:, feature_cols]).to_numpy(dtype=float)

# Input normalization
for i in range(X.shape[1]):
    avg = np.mean(X[:, i])
    stddev = np.std(X[:, i])
    X[:, i] = (X[:, i] - avg) / stddev

print(X.shape)

# Our target variable is Sex
sex = df.pop('Sex').values

# convert to negative=-1 and positive=1 using functions from scikit-learn
label_encoder = LabelEncoder()
```

```
y = label_encoder.fit_transform(sex)
y = np.where(y==0, -1, y)
print(y.shape)
print(min(y), max(y))

#       - training,
#       - printing the final weight vector
np.random.seed(567)

w_logistic, loss = train_logistic(X, y, num_iterations = 5000, return_history =␣
 ↪True)


print()
print("Final weight:\n", w_logistic)
print()
print("Final loss:\n", loss[-1])


#       - computing the prediction and getting the misclassified points

y_pred = predict_logistic(X, w_logistic)

threshold = 0.5
y_class = define_class(y_pred, threshold)

misclassified = np.where(y != y_class)[0]

print(f'\nWe have {misclassified.size} misclassified points using {threshold}␣
 ↪as threshold.')

#       - plotting the loss curve

plt.figure(figsize = (12, 8))
plt.plot(loss)
plt.xlabel('Iteration #')
plt.ylabel('Cross Entropy Loss')
plt.show()
```

```
Using features = ['Height', 'Weight', 'Shoe number', 'Age']

(130, 4)
(130,)
-1 1

Final weight:
 [ 2.20707971 -1.28294485 -0.7653262   5.43646254 -0.50329359]
```

```
Final loss:
 0.2217107637035728

We have 12 misclassified points using 0.5 as threshold.
```



We can see that in this example that the bigger the d, the better the prediction.

6 4. MNIST Dataset

This is a well known dataset, commonly used as a first example to illustrate image classification tasks. We could say it is the "Hello world!" of image classification. It consists of handwritten digits, divided into 60000 training images and 10000 test images. All images are gray-scale (one channel with pixel intensities varying from 0 to 255) and have size 28×28. There are 10 classes, corresponding to digits 0 to 9.

We could use the 28×28 pixel intensities as features. However, here we will perform feature extraction from the images and then the classification based on the extracted features.

The dataset is available in many places. Here we will use the one available with Keras 1. More information on MNIST can be found at the oficial site.

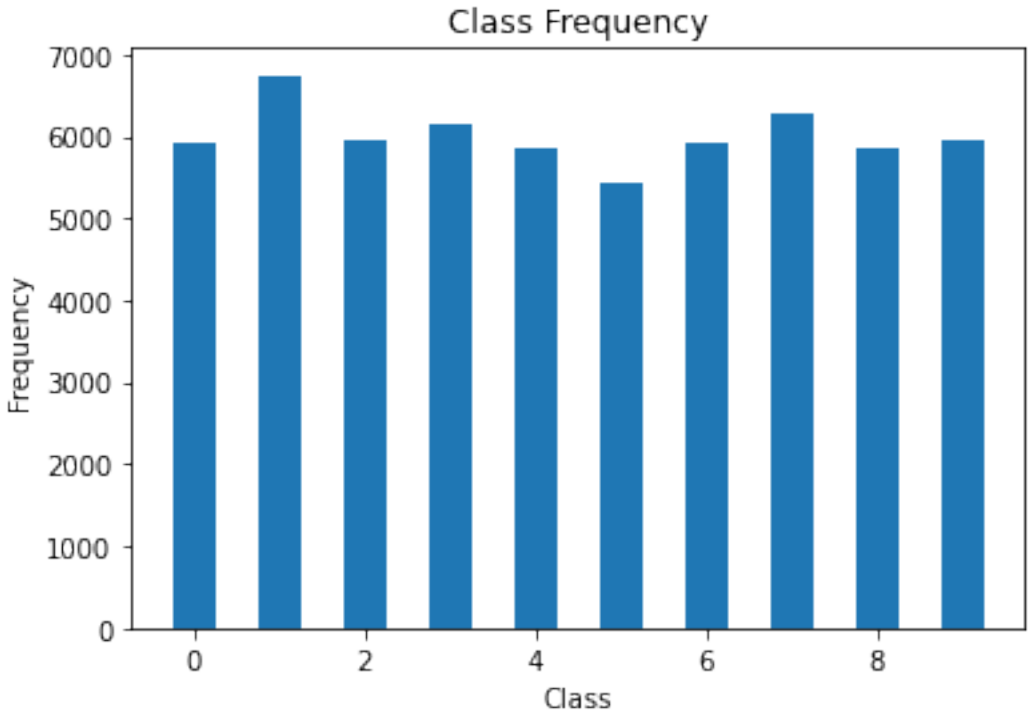
6.1 4.1. Getting and inspecting the data

[22]:
```
(X_train_all, y_train_all), (X_test_all, y_test_all) = mnist.load_data()

print(X_train_all.shape, y_train_all.shape)
print(X_test_all.shape, y_test_all.shape)
```

```
(60000, 28, 28) (60000,)
(10000, 28, 28) (10000,)
```

Class distribution of MNIST (training set)

[23]:
```
unique, counts = np.unique(y_train_all, return_counts=True)
plt.bar(unique, counts, 0.5)
plt.title('Class Frequency')
plt.xlabel('Class')
plt.ylabel('Frequency')
plt.show()
```



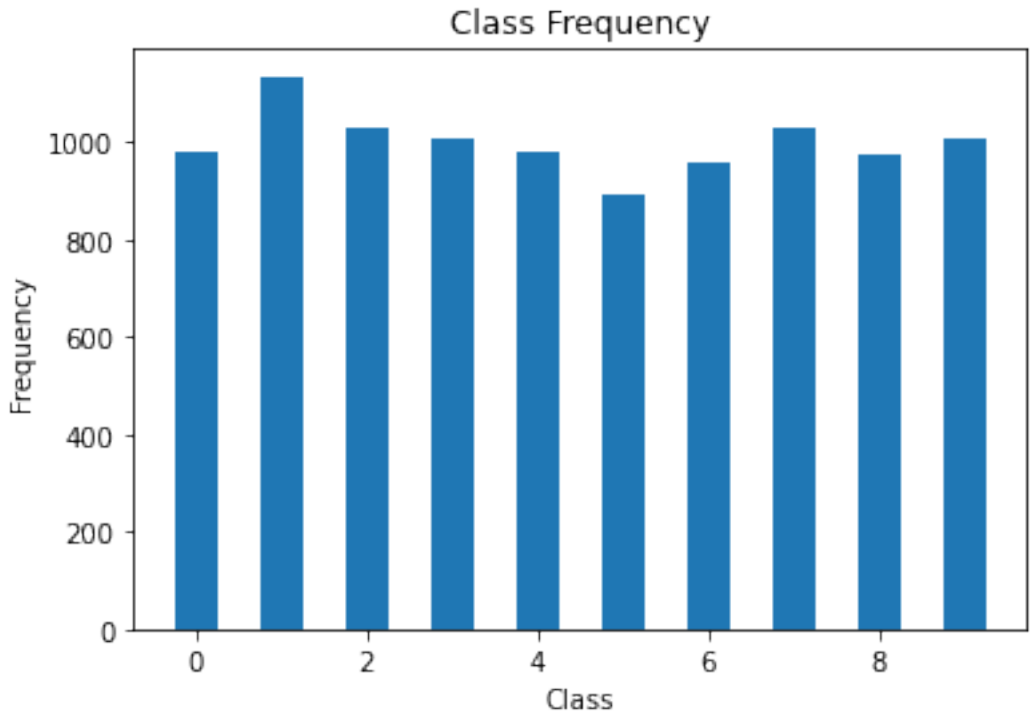
6.1.1 Class distribution of MNIST (test set)

Repeat **Class distribution of MNIST** for the testing set (next cell) and compare and comment about the distributions of the training and of the testing sets. Do you think this type of comparison is important? Comment.

===> Your comments here

First, we can see that the train dataset is balanced. And also, we can see that the distributions in both train and test set are similar. These two informations are very important to analyse, the first one because it is important to be balanced otherwise we could not be doing well with the smaller classes, and the second one is also good so we know that we are training the model with informations very similar to the real world.

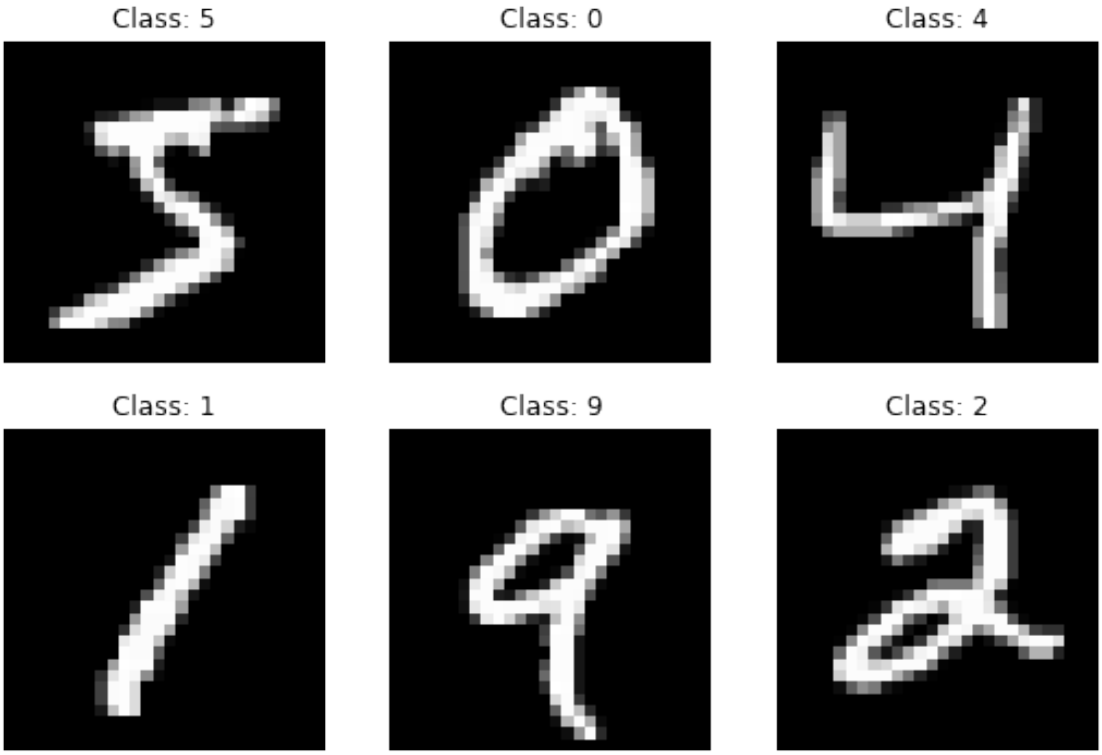
[24]:
```
# ===> Your code here
unique, counts = np.unique(y_test_all, return_counts=True)
plt.bar(unique, counts, 0.5)
plt.title('Class Frequency')
plt.xlabel('Class')
plt.ylabel('Frequency')
plt.show()
```



Visualization of some of the examples

[25]:
```
fig, ax = plt.subplots(2, 3, figsize = (9, 6))

for i in range(6):
    ax[i//3, i%3].imshow(X_train_all[i], cmap='gray')
    ax[i//3, i%3].axis('off')
    ax[i//3, i%3].set_title("Class: %d"%y_train_all[i])
```

```
plt.show()
```



6.2 4.2. Feature extraction

Note that the images consist of 28×28=784 pixel values and they could be used as "raw features" of the images. However, here we will extract some features from the images and perform classification using them instead of the pixel values.

6.2.1 Mean intensity

In the book *Learning from Data* 2, one of the attributes (features) used by the authors is the mean intensity of the pixel values. This feature is directly related to the proportion of the pixels corresponding to the digit in the image. For instance, it is reasonable to expect that a digit 5 or 2 occupies more pixels than the digit 1 and, therefore, the mean intensity of the first two should be larger than that of the digit 1.

6.2.2 Symmetry

The second attribute used by the authors is horizontal symmetry.

Symmetry will be defined in terms of asymetry. We define asymmetry as the pixelwise mean of the absolute difference between the pixels values from the original image and those from the corresponding horizotally flipped image. Then, symmetry is defined as the negative of asymmetry.

[26]:
```
def mean_intensity(image):
    return np.mean(image)

def Hsimmetry(image):
    # The processing below invert the order of the columns of the image
    reflected_image = image[:, ::-1]
    return -np.mean(np.abs(image - reflected_image))
```

6.2.3 Pixels → Features

The above functions for feature extraction will be applied to the samples, both on the training and the test sets. After the feature extraction process below, each image will be represented by two features.

[27]:
```
# Function that converts an image into a list of features,
# using the feature computation functions defined above
def convert2features(image):
    return np.array([mean_intensity(image),
                     Hsimmetry(image)])

# feature names
F = ['Mean intensity', 'Hsimmetry']

# Generate the feature representation for all images
X_train_features = np.array([convert2features(image) for image in X_train_all])
X_test_features  = np.array([convert2features(image) for image in X_test_all])

print(X_train_features.shape)
print(X_test_features.shape)

for i in range(0,X_train_features.shape[1]):
    print()
    print("Mean of '%s' = %f" \
          %(F[i],np.mean(X_train_features[:, i])))
```

```
(60000, 2)
(10000, 2)

Mean of 'Mean intensity' = 33.318421

Mean of 'Hsimmetry' = -32.617796
```

Normalization of feature values is a common procedure. Here we apply the z-score formula. Note that the normalization parameters (mean and standard deviation) are computed only on training data. To normalize the test data, we use the same parameters. (We suggest you to think why we should not compute the mean and standard deviation over the training+test set. There is no need to answer this here.)

[28]:
```
# Adjust the scale of feature values; standardize them.
# (Yes, the features in the test set should be standardized using
#  the statistics of the features in the training set) -- why ??
for i in range(X_train_features.shape[1]):
    avg = np.mean(X_train_features[:, i])
    stddev = np.std(X_train_features[:, i])
    X_train_features[:, i] = (X_train_features[:, i] - avg) / stddev
    X_test_features[:, i] = (X_test_features[:, i] - avg) / stddev

print("Shape of X_train: ", X_train_features.shape)
print("Shape of X_test: ", X_test_features.shape)

# print the mean value of each of the features
print()
print("Training set, after normalization:")
for i in range(0,X_train_features.shape[1]):
    print("  Mean value of '%s' = %f" \
          %(F[i],np.mean(X_train_features[:, i])))

print()
print("Testing set, after normalization:")
for i in range(0,X_test_features.shape[1]):
    print("  Mean value of '%s' = %f" \
          %(F[i],np.mean(X_test_features[:, i])))
```

```
Shape of X_train:  (60000, 2)
Shape of X_test:  (10000, 2)

Training set, after normalization:
  Mean value of 'Mean intensity' = -0.000000
  Mean value of 'Hsimmetry' = -0.000000

Testing set, after normalization:
  Mean value of 'Mean intensity' = 0.042822
  Mean value of 'Hsimmetry' = 0.020592
```

6.3 4.3 Logistic regression training and testing

6.3.1 4.3.1 Select a subset from two of the classes

Here we select two classes as well as a subset of the examples in each class. All code from here on will use the selected subset. You may change later the selected classes and the number of samples in each class.

First, let us select two classes

[29]:
```
P = 5  # positive class
N = 1  # negative class
```

```
X_train_P = X_train_features[y_train_all == P]
X_train_N = X_train_features[y_train_all == N]
y_train_P = y_train_all[y_train_all == P]
y_train_N = y_train_all[y_train_all == N]

X_test_P = X_test_features[y_test_all == P]
X_test_N = X_test_features[y_test_all == N]
y_test_P = y_test_all[y_test_all == P]
y_test_N = y_test_all[y_test_all == N]

print("Positive class: ", X_train_P.shape, y_train_P.shape)
print("Negative class: ", X_train_N.shape, y_train_N.shape)
```

```
Positive class:  (5421, 2) (5421,)
Negative class:  (6742, 2) (6742,)
```

6.3.2 Now we will select a subset of examples from each of the two classes

In the following cell, write where it is indicated by ===> the code to change the label of the positive class to +1 and of the negative class to -1

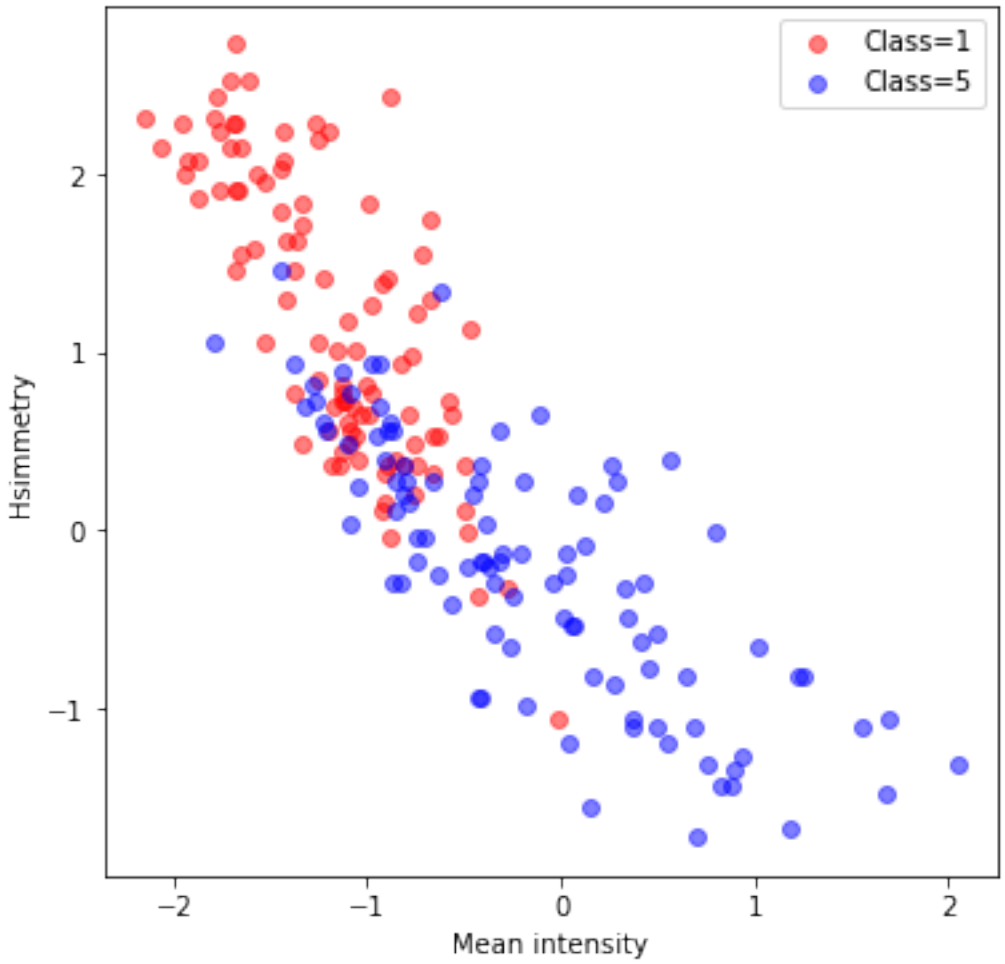
[30]:
```
# Number of positives and negatives to be effectively considered
# in the training data to be explored in the remainder of this notebook
nP = 100
nN = 100

X_train = np.concatenate([X_train_P[:nP], X_train_N[:nN]], axis = 0)
y_train = np.concatenate([y_train_P[:nP], y_train_N[:nN]], axis = 0).
 ↪astype('float32')

X_test = np.concatenate([X_test_P, X_test_N], axis = 0)
y_test = np.concatenate([y_test_P, y_test_N], axis = 0).astype('float32')


# ===> Change positive class label to +1 and negative class label to -1

def change_label_class(y, P):
    changer = lambda t: 1 if t == P else -1
    vfunc = np.vectorize(changer)
    return vfunc(y)

y_train = change_label_class(y_train, P)
y_test = change_label_class(y_test, P)

### Your code insert ends here

# Shuffle
np.random.seed(56789)
```

```
def shuffle(X, y):
    # input and output must be shuffled equally
    perm = np.random.permutation(len(X))
    return X[perm], y[perm]

X_train, y_train = shuffle(X_train, y_train)
X_test, y_test = shuffle(X_test, y_test)

print("Training X and y --> ", X_train.shape, y_train.shape)
print()
print("Testing X and y --> ", X_test.shape, y_test.shape)
```

```
Training X and y -->  (200, 2) (200,)

Testing X and y -->  (2027, 2) (2027,)
```

Plot the selected data Let us plot the selected subset of data. Negative examples will be plotted in red and positive ones in blue.

[31]:
```
def plot_features(ax,X,y):
    # negatives in red
    ax.scatter(X[y==-1,0], \
               X[y==-1,1], \
               label='Class=%d'%N, c = 'red', alpha = 0.5)

    # and positives in blue
    ax.scatter(x=X[y==1,0], \
               y=X[y==1,1], \
               label='Class=%d'%P, c = 'blue', alpha = 0.5)

    ax.set_xlabel(F[0])
    ax.set_ylabel(F[1])
    ax.legend(loc='best')

fig = plt.figure(figsize=(6,6))
ax = fig.add_subplot(111)
plot_features(ax,X_train,y_train)
plt.show()
```



6.3.3 4.3.2 Training

Run the code in the following cell a few times, each time with different values for the learning rate and the number of iterations. Comment the behavior of the loss curve. Which values do you consider as good choices?

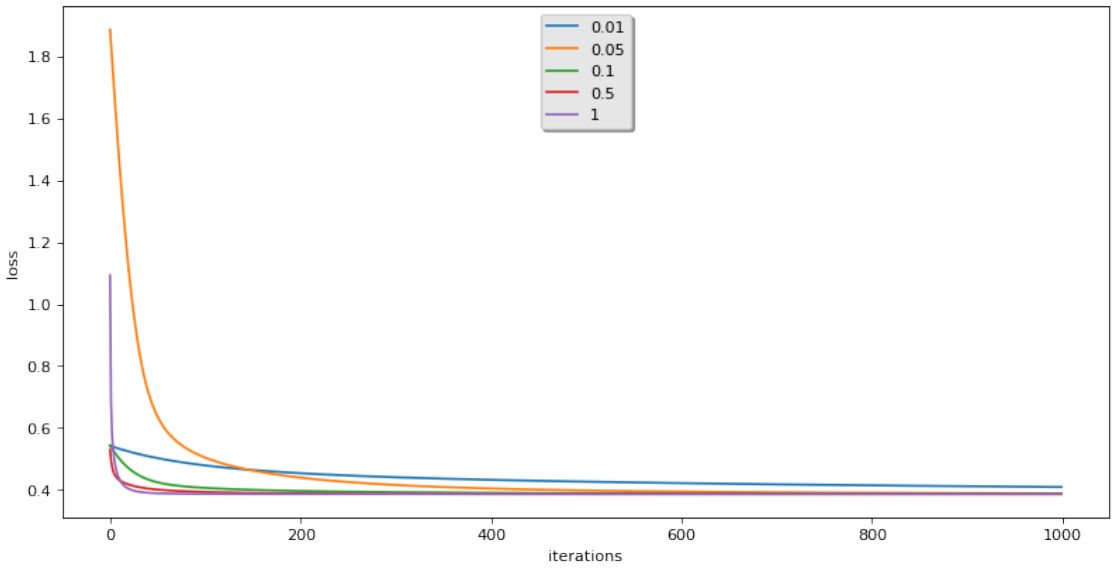
Note: for your submission, keep the execution output corresponding to the best parameter values you have found.

===> Your comments here

We can see that if we have a small num_iterations, we stop too early and don't get the best we could. Although, we can see that there is a point were we have no improvements, so it's not worth having very high values to num_iterations. About the learning_rate, we can see that if we have bigger values, we find a stable value faster. First I thought we should't use very big learning_rate, but in this example even using learning_rate = 1 we have the same output as 0.005, but much faster. With learning_rate = 1 , we can use num_iterations = 1000 and we already have a stable value, and if we add more iterations we have almost no improvements. We kept the plot with

the chosen configuration and also a plot comparing the loss curve of some learning rates with num_iterations = 1000 because it is very interesting.

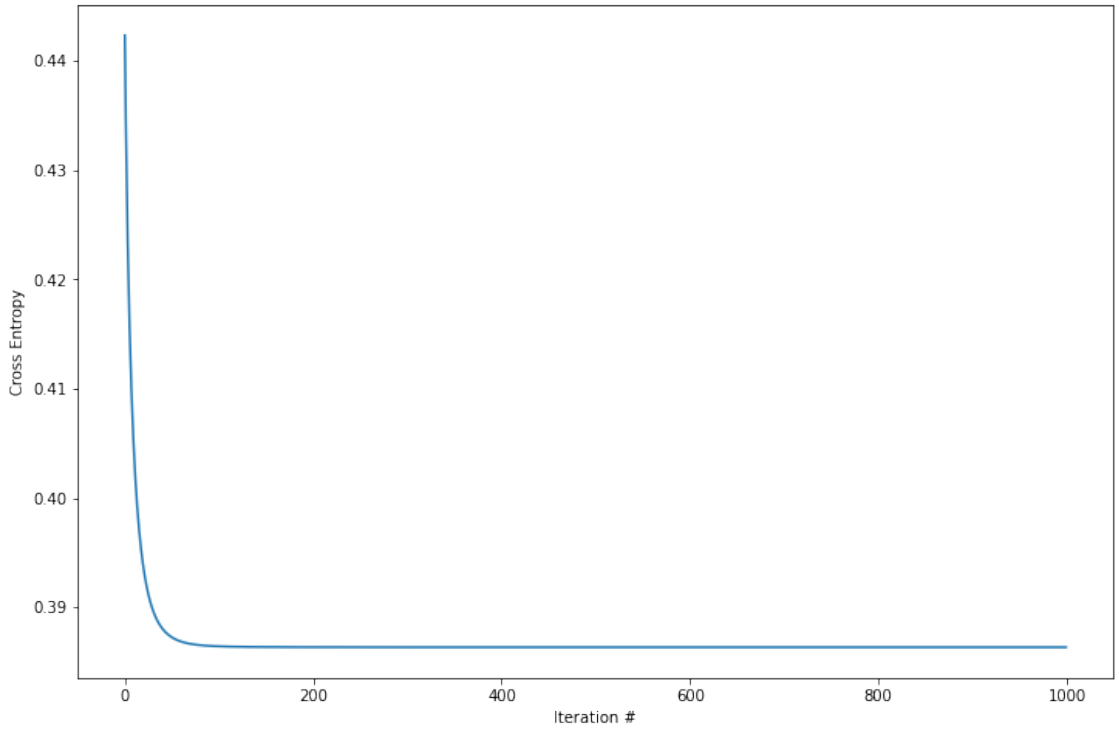
[32]:
```
## Comparting different learning rates
learning_rates = [0.01, 0.05, 0.1, 0.5, 1]
loss = {}
l = {}
plt.figure(figsize=(12, 6), dpi=80)
for i in learning_rates:
    l[i] = i
    _, loss[i] = train_logistic(X_train, y_train,\
                                    learning_rate = i,
                                    num_iterations = 1000,\
                                    return_history = True)

    plt.plot(np.squeeze(loss[i]), label= str(l[i]))


plt.ylabel('loss')
plt.xlabel('iterations')

legend = plt.legend(loc='upper center', shadow=True)
frame = legend.get_frame()
frame.set_facecolor('0.90')
plt.show()
```



[33]:
```
## Final configuration
```

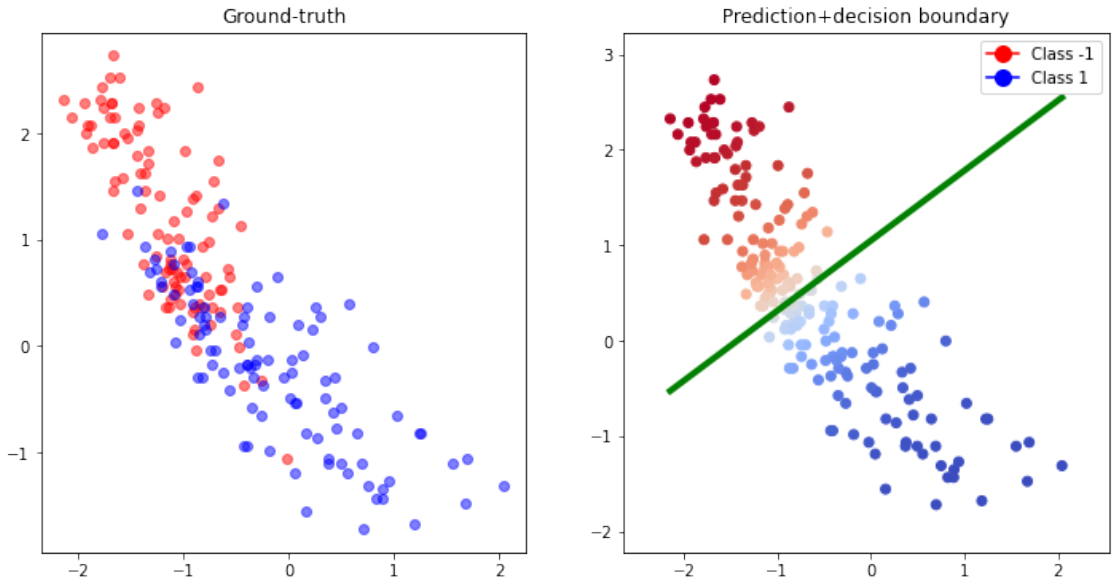
```
np.random.seed(56789)
w_logistic, loss = train_logistic(X_train, y_train,\
                                  learning_rate = 1,
                                  num_iterations = 1000,\
                                  return_history = True)

print()
print("Final weight:\n", w_logistic)
print()
print("Final loss:\n", loss[-1])

plt.figure(figsize = (12, 8))
plt.plot(loss)
plt.xlabel('Iteration #')
plt.ylabel('Cross Entropy')
plt.show()
```

```
Final weight:
 [ 1.80481717  1.26430913 -1.72312858]

Final loss:
 0.38633952465344307
```



Ploting the scores and decision boundary graphs (training set)

[34]:
```
x1min = min(X_train[:,0])
x1max = max(X_train[:,0])
x2min = min(X_train[:,1])
x2max = max(X_train[:,1])

y_pred = predict_logistic(X_train, w_logistic)

fig = plt.figure(figsize=(12,6))
ax1 = fig.add_subplot(121)
ax1.set_title("Ground-truth")

# plot negatives in red
ax1.scatter(X_train[y_train==-1,0], \
            X_train[y_train==-1,1], \
            alpha = 0.5, \
            c = 'red')

# and positives in blue
ax1.scatter(x=X_train[y_train==1,0], \
            y=X_train[y_train==1,1], \
            alpha = 0.5, \
            c = 'blue')

ax2 = fig.add_subplot(122)
ax2.set_title("Prediction+decision boundary")

ax2.scatter(x = X_train[:,0], y = X_train[:,1], c = -y_pred, cmap = 'coolwarm')
ax2.legend(handles=legend_elements, loc='best')

ax2.set_xlim([x1min-0.5, x1max+0.5])
ax2.set_ylim([x2min-0.5, x2max+0.5])

p1 = (x1min, -(w_logistic[0] + (x1min)*w_logistic[1])/w_logistic[2])
p2 = (x1max, -(w_logistic[0] + (x1max)*w_logistic[1])/w_logistic[2])

lines = ax2.plot([p1[0], p2[0]], [p1[1], p2[1]], '-')
plt.setp(lines, color='g', linewidth=4.0)

plt.show()
```



6.3.4 Confusion matrix (training set)

Recall that the logistic regression returns a score $\hat{p}$ in $[0, 1]$, which can be interpreted as the probability $p(y = 1|\mathbf{x})$. To compute the confusion matrix, one needs to choose a threshold value T to decide the final class label (that is $\hat{y} = 1 \iff \hat{p} \geq T$).

Play with the threshold value in the code (following cell). Did you manage to find a threshold value (other than 0.5) that improves accuracy? How threshold relates to TP, FP, TN and FN ? Comment.

===> Your comment here

To find the best threshold I tested a number of options and I got the best one according to the accuracy. We can see that to have bigger accuracy, we want bigger (TP+TN), that is, TP and TN big at the same time, it's no use if only one is big. Hence FP and FN smaller as they are complements.

[35]:
```
def get_accuracy(y, y_pred):
    TP = np.sum((y_pred == 1) * (y == 1))
    TN = np.sum((y_pred == -1) * (y == -1))

    FP = np.sum((y_pred == 1) * (y == -1))
    FN = np.sum((y_pred == -1) * (y == 1))

    total = TP+FP+TN+FN
    accuracy = (TP+TN)/total
    return accuracy

## Test a number of thresholds and get the best one according to the accuracy
```

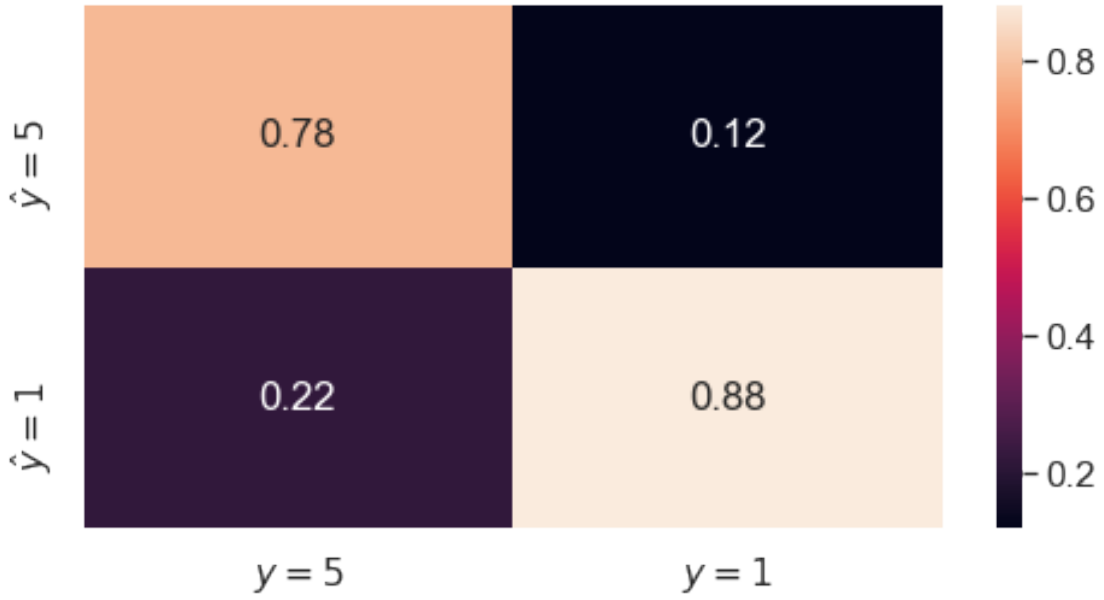
```
threshold = [0.1, 0.2, 0.3, 0.4, 0.5, 0.55, 0.6, 0.65, 0.7, 0.8, 0.9]

max_accuracy = -1
best_t = -1

for t in threshold:
    p_hat = predict_logistic(X_train, w_logistic)
    y_hat = np.where(p_hat > t, 1, -1)

    accuracy = get_accuracy(y_train, y_hat)
    if accuracy > max_accuracy:
        max_accuracy = accuracy
        best_t = t

print(f"The best threshold is = {best_t} with accurancy = {max_accuracy}.\n")
```

```
The best threshold is = 0.55 with accurancy = 0.83.
```

[36]:
```
def plot_confusion_matrix(y, y_pred):
    """
    It receives an array with the ground-truth (y)
    and another with the prediction (y_pred), both with binary labels
    (positive=+1 and negative=-1) and plots the confusion
    matrix.
    It uses P (positive class id) and N (negative class id)
    which are "global" variables ...
    """
    TP = np.sum((y_pred == 1) * (y == 1))
    TN = np.sum((y_pred == -1) * (y == -1))

    FP = np.sum((y_pred == 1) * (y == -1))
    FN = np.sum((y_pred == -1) * (y == 1))

    total = TP+FP+TN+FN
    print("TP = %4d    FP = %4d\nFN = %4d    TN = %4d"%(TP,FP,FN,TN))
    print("Accuracy = %d / %d (%f)\n" %((TP+TN),total, (TP+TN)/total))
    confusion = [
        [TP/(TP+FN), FP/(TN+FP)],
        [FN/(TP+FN), TN/(TN+FP)]
    ]

    df_cm = pd.DataFrame(confusion, \
                         ['$\hat{y} = %d$'%P, '$\hat{y} = %d$'%N],\
                         ['$y = %d$'%P, '$y = %d$'%N])
    plt.figure(figsize = (8,4))
    sb.set(font_scale=1.4)
```

```
    sb.heatmap(df_cm, annot=True) #, annot_kws={"size": 16}, cmap = 'coolwarm')
    plt.show()


threshold = 0.55

p_hat = predict_logistic(X_train, w_logistic)
y_hat = np.where(p_hat > threshold, 1, -1)

total = len(y_hat)

plot_confusion_matrix(y_train, y_hat)
```

```
TP =   78    FP =   12
FN =   22    TN =   88
Accuracy = 166 / 200 (0.830000)
```



6.3.5 4.3.3. Testing

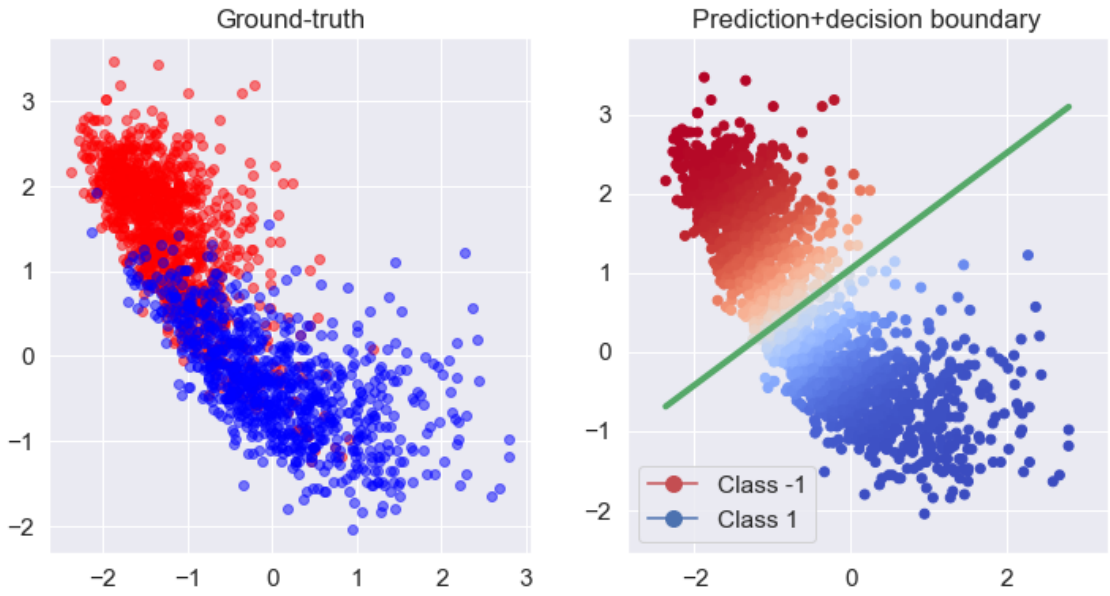
Now that you have trained the algorithm, let us evaluate its performance on the test set. Plot the scatter plot graphs (as above) and the confusion matrix (using threshold=0.5). Do you think the algorithm is doing a good generalization? Comment.

===> Your comment here

The algorithm is doing a very good generalization, since we can see a good accuracy (0.88) in the test set. Analysing the scatter plot, we can see that there is a lot of misclassified points, but we

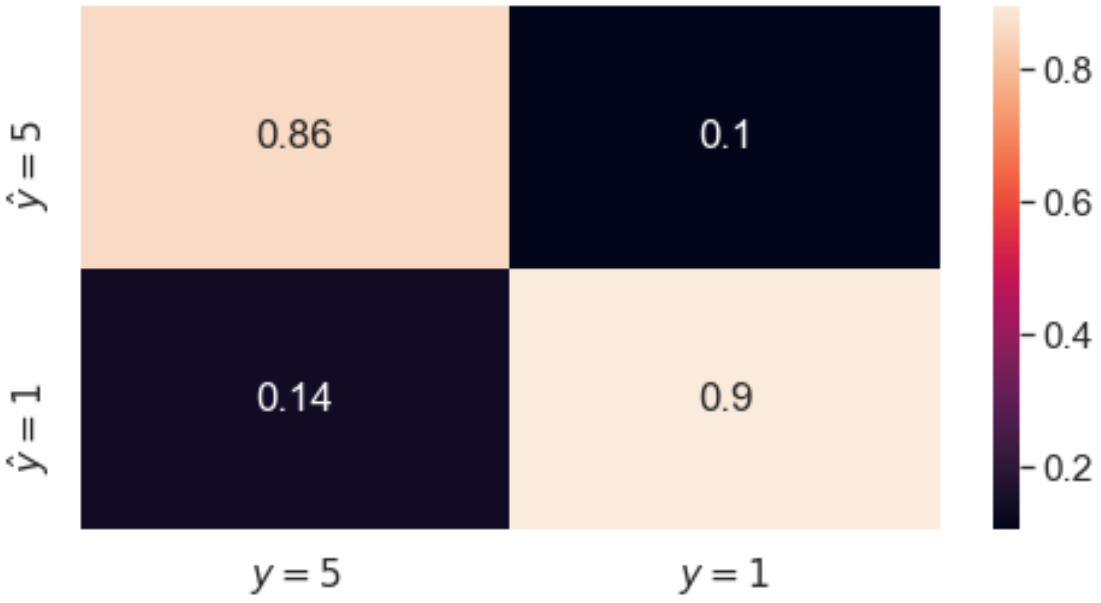
also can see that are overlapping points, so we would need more dimensions to separate them.

[37]:
```
# ===> Your code for the scatter plot (test set)

x1min = min(X_test[:,0])
x1max = max(X_test[:,0])
x2min = min(X_test[:,1])
x2max = max(X_test[:,1])

y_pred = predict_logistic(X_test, w_logistic)

fig = plt.figure(figsize=(12,6))
ax1 = fig.add_subplot(121)
ax1.set_title("Ground-truth")

# plot negatives in red
ax1.scatter(X_test[y_test==-1,0], \
            X_test[y_test==-1,1], \
            alpha = 0.5, \
            c = 'red')

# and positives in blue
ax1.scatter(x=X_test[y_test==1,0], \
            y=X_test[y_test==1,1], \
            alpha = 0.5, \
            c = 'blue')

ax2 = fig.add_subplot(122)
ax2.set_title("Prediction+decision boundary")

ax2.scatter(x = X_test[:,0], y = X_test[:,1], c = -y_pred, cmap = 'coolwarm')
ax2.legend(handles=legend_elements, loc='best')

ax2.set_xlim([x1min-0.5, x1max+0.5])
ax2.set_ylim([x2min-0.5, x2max+0.5])

p1 = (x1min, -(w_logistic[0] + (x1min)*w_logistic[1])/w_logistic[2])
p2 = (x1max, -(w_logistic[0] + (x1max)*w_logistic[1])/w_logistic[2])

lines = ax2.plot([p1[0], p2[0]], [p1[1], p2[1]], '-')
plt.setp(lines, color='g', linewidth=4.0)

plt.show()
```



[38]:
```
# ===> your code for the confusion matrix (test set)

threshold = 0.5

p_hat = predict_logistic(X_test, w_logistic)
y_hat = np.where(p_hat > threshold, 1, -1)

plot_confusion_matrix(y_test, y_hat)
```

```
TP =  768    FP =  118
FN =  124    TN = 1017
Accuracy = 1785 / 2027 (0.880612)
```



6.4 Extra (optional)

- If you got to this point, make a copy of your notebook. Run the copy notebook changing the number of positive and negative examples in the MNIST case. Try an unbalanced training set and observe if there are any effects in the accuracy on the test set. Additionaly, you may try with a different pair of classes. If you wish, you can summarize HERE the experiments you did and comment whatever you found interesting. There is no need to submit the copy notebook.
- You can also compare the results obtained with your algorithm with the ones generated by a standard implementation like the one in the scikit-learn library

===> Your comments here

First of all, trying the same classes but with more data, still balanced, using 5000 examples of each class. We got 0.8668 of accuracy on train with threshold = 0.4 and 0.87 of accuracy on test. We could see that the behavior was very similar on test than the one with 100 examples.

Second, trying to use unbalanced data, with 5000 examples of the positive class (5) and 100 of the negative (1) on train. We got excelent accuracy on train (0.9868) with threashold = 0.65, but with high TP but low TN, and on the test set we got just 0.727 of accuracy, worse than when we trained with balanced data (0.88).

Finally, changing the positive class to 8 and negative to 1, as these seem to be less similar numbers, using 100 examples of each class, we could find 0.905 of accuracy in the training set and 0.92 of accuracy in the test set, with threshold = 0.5.

6.4.1 Remarks

The organization of the code in this notebook does not follow the principles of good programming. For instance, a same piece of code is used repeatedly in several places, the same variable name is used to keep data of different nature, and so on.