Assignment 6: Logistic Regression

Copyright and Fair Use

Automatic Testing Guidelines

Automatic unittesting requires you to submit a notebook which contains strictly defined objects. Strictness of definition consists of unified shapes, dtypes, variable names and more.

Within the notebook, we provide detailed instruction which you should follow in order to maximise your final grade.

Name your notebook properly, follow the pattern in the template name:

Assignment_N_NameSurname_matrnumber

1. N - number of assignment
2. NameSurname - your full name where every part of the name starts with a capital letter, no spaces
3. matrnumber - you student number on ID card (without k, potenitially with a leading zero)

Don't add any cells but use the ones provided by us. You may notice that all cells are tagged such that the unittest routine can recognise them. Before you sumbit your solution, make sure every cell has its (correct) tag!

You can implement helper functions where needed unless you put them in the same cell they are actually called. Always make sure that implemented functions have the correct output and given variables contain the correct data type. In the descriptions for every function you can find information on what datatype an output should have and you should stick to that in order to minimize conflicts with the unittest. Don't import any other packages than listed in the cell with the "imports" tag.

Questions are usually multiple choice (except the task description says otherwise) and can be answered by changing the given variables to either "True" or "False". "None" is counted as a wrong answer in any case!

Note: Never use variables you defined in another cell in your functions directly; always pass them to the function as a parameter. In the unitest, they won't be available either. If you want to make sure that everything is executable for the unittest, try executing cells/functions individually (instead of running the whole notebook).

In [1]:
```
import numpy as np
from sklearn.metrics import roc_curve, auc
import matplotlib
import matplotlib.pyplot as plt
```

Task 1:

The goal of this exercise is to implement logistic regression from scratch using only numpy. Start with the following tasks:

- Implement the formula for the gradient computed in the lecture. In particular you should implement a function

`logistic_gradient(w, x, y)` that takes a parameter vector $\mathbf{w}$, a data matrix $\mathbf{X}$ and a label vector $\mathbf{y}$ and returns the gradient $\frac{\partial L}{\partial \mathbf{w}}$, where L is the negative log-likelihood for the Bernoulli distribution, i.e. the cross-entropy loss.

- Implement a function `cost(w, x, y)`, that takes the same parameters but returns the cross-entropy loss.
- Test whether the gradient calculated by `logistic_gradient(w, x, y)` is correct via Gradient Checking. To do so, implement

a function `numerical_gradient(w, x, y)` that takes the same parameters as `logistic_gradient`, but calculates the gradient numerically via the central difference quotient, using $\epsilon = 10^{-4}$ as suggested in the lecture slides.

- Implement the function `generate_random(nr_samples, nr_features)` that generates a random data matrix consisting of 5 data points with 10 features drawn from a standard normal distribution as well as corresponding random binary labels and a random weight vector, whose entries again stem from the standard normal distribution. Hint: to generate the distributions use `np.random.normal` and `np.random.randint`.
- Implement the function `comparison(grad_a,grad_n)` that takes the analytical and the numerical gradient as inputs respectively. The function should check whether the two vectors deviate more than $\epsilon = 10^{-7}$ or not from each other (they shouldn't ;))

Code 1.1 (5 points):

In [2]:
```
"""
Function that computes the logistic gradient
@param w, np array, weights
@param x, np array, data matrix
@param y, np array, data labels

@output gradient, np array, gradient vector
"""
def logistic_gradient(w, x, y):
    gradient=0
    for index in range(x.shape[0]):
        w_transpose=np.transpose(w)
        sigmoid=1/(1+np.exp((-1)*np.dot(w_transpose,x[index])))
        gradient=gradient+x[index]*(sigmoid-y[index])
    return gradient
```

Code 1.2 (5 points):

In [3]:
```
"""
Function that computes the cross-entropy loss
@param w, np array, weights
@param x, np array, data matrix
@param y, np array, data labels

@output loss, float, cross-entropy loss
"""
def cost(w, x, y):
    loss=0
    for index in range(x.shape[0]):
        w_transpose=np.transpose(w)
        sigmoid=1/(1+np.exp((-1)*np.dot(w_transpose,x[index])))
        loss=loss+y[index]*np.emath.log(sigmoid)+(1-y[index])*np.emath.log(1-sigmoid)
    loss=loss*(-1)
    return loss
```

Code 1.3 (10 points):

In [4]:
```
"""
Function that computes the numerical gradient
@param w, np array, weights
@param x, np array, data matrix
@param y, np array, data labels

@output dw, np array, numerical gradient
"""
def numerical_gradient(w, x, y):
    dw=np.zeros(w.shape)
    quotient=10**(-4)
    for index in range(x.shape[1]):
        e=np.zeros(w.shape)
        e[index]=1
        num_gradient=(cost(w+quotient*e,x,y)-cost(w-quotient*e,x,y))/(2*quotient)
        dw[index]=num_gradient
    return dw
```

Code 1.4 (10 points):

In [5]:
```
"""
Function that generates a random matrix X and the random vectors y and weights
@param nr_samples, int, the number of samples you should generate
@param nr_features, int, the number of feature each sample has

@output X_random, np array, random samples
@output y_random, np array, random targets
@output w_random, np array, random weights
"""
def generate_random(nr_samples, nr_features):
    X_random=np.random.normal(size=(nr_samples,nr_features))
    w_random=np.random.normal(size=nr_features)
    y_random=np.random.randint(2,size=nr_samples)
    return X_random, y_random, w_random
```

Code 1.5 (10 points):

In [6]:
```
"""
Function that compares two array
@param grad_a, np array, the analytical gradient
@param grad_n, np array, the numberical gradient

@output close, bool , True if the arrays are similar, False if they are not
"""
def comparison(grad_a,grad_n):
    deviation=10**(-7)
    close=np.allclose(grad_a,grad_n,deviation)
    return close
```

In [7]:
```
#Nothing to do here, if you did everything correctly you can just run this code and should see the correct results
n = 5
d = 10
X_random, y_random, w_random = generate_random(n,10)
analytical_gradient = logistic_gradient(w_random,X_random,y_random)
num_gradient = numerical_gradient(w_random,X_random,y_random)
comparison_result = comparison(analytical_gradient, num_gradient)
print("X =",X_random,"\n")
print("y =",y_random,"\n")
print("w = ",w_random,"\n")
print("Logistic gradient:\n",analytical_gradient,"\n")
print("Numerical gradient:\n", num_gradient, "\n")
print("Vectors within absolute tolerance of 10^-7: ",comparison_result)
```

```
X = [[-0.17298956 -1.79765965 -0.62673364  0.77924542  0.24781919 -0.8614072
   0.64987305  0.33637928 -0.82693698  1.03563463]
 [-0.41862369  0.76893242 -1.10553226  2.75726974  0.27433505  1.70942912
   1.09473921  0.18213744  0.50662855 -1.15605147]
 [ 0.72485706  1.1088547  -0.03481415  0.86537631 -0.14560743 -0.7255363
   0.10070181 -0.04810799  1.58356374  0.1377811 ]
 [ 1.19835329  0.81352261 -0.30963375 -0.68594083  0.65123053 -1.12956795
   0.7629413   1.97475433 -0.80140995  1.26439161]
 [-0.93400266 -0.71012339  0.96881804 -1.2429211   0.76774434 -1.70432563
   0.34810329  0.21482947 -2.39968536  1.6738006 ]]

y = [1 1 0 1 1]

w =  [ 0.2987919  -1.59272413  0.58873476 -0.0776144  -0.06881164  1.393102
 -0.42726713  0.33280158 -0.91871352  0.04975432]

Logistic gradient:
 [-0.46682603 -0.77743219  0.89465882 -1.15595718 -0.92712189  0.23374846
 -1.54524768 -1.84928748  0.96795031 -0.75189723]

Numerical gradient:
 [-0.46682603 -0.77743219  0.89465882 -1.15595717 -0.92712189  0.23374846
 -1.54524768 -1.84928747  0.96795031 -0.75189723]

Vectors within absolute tolerance of 10^-7:  True
```

Next we intend to apply logistic regression on a real data set.

- Implement a function `fitLogRegModel(x_train, y_train, eta=1e-4, max_iter=1e5)` that uses Logistic Regression with Gradient Descent to train classifiers on the training set. Use randomly initialized weights, drawn from a uniform distribution between -1 and 1, a learning rate η (eta) of 10^{-4} and a maximum number of iterations of $1e5$. Furthermore the algorithm should stop if the difference between the loss of the last iteration step and the current loss is less than η. Store all the losses in a list to have some insights in the learning procedure later on. Also print the losses in 1000 step intevals. The function should return the model weights and the list containing all the losses.
- Furthermore, implement a function `predictLogReg(w, x)` that returns the prediction for the given parameter vector $\mathbf{w}$ and feature vector $\mathbf{x}$.

Hint: for intialization use `np.random.uniform`.

Code 1.6 (25 points)

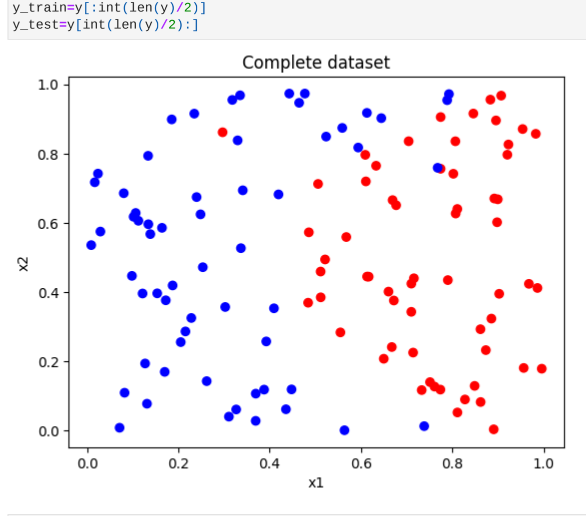
In [8]:
```
"""
Function that fits a logistic regression model to given dat
@param x_train, np array, training data
@param y_train, np array, training samples

@output w, np array , the final weight array
@output losses, list , list holding all the losses from the training (including the loss before the training)
"""
def fitLogRegModel(x_train, y_train, eta=1e-4, max_iter=100000):
    w=np.random.uniform(-1,1,x_train.shape[1])
    losses=[]
    previous_loss=cost(w,x_train,y_train)
    losses.append(previous_loss)
    for it in range(max_iter):
        gradient=logistic_gradient(w,x_train,y_train)
        w=w-eta*gradient
        current_loss=cost(w,x_train,y_train)

        losses.append(current_loss)
        if it%1000==0:
            print(current_loss)

        if abs(current_loss-previous_loss)<eta:
            break
        previous_loss=current_loss

    return w,losses
```

Code 1.7 (5) points)

In [9]:
```
"""
Function that calculates the prediction for one or more new samples
@param w, np array, weights
@param x, np array, samples for inference

@output prediction, np array, the calculated predictions
"""
def predictLogReg(w, x):
    prediction=1/(1+np.exp((-1)*np.dot(x,w)))
    return prediction
```

Now we fit the logistic regression model from above to the training data and print the parameters for the test data.

In [10]:
```
#nothing to do here

from sklearn.utils import shuffle
# Read data, split into X(features) and y(labels)
Z = np.genfromtxt('DataSet_LR_a.csv', delimiter=',',skip_header=1)
X, y = Z[:,:-1], Z[:,-1]
X = np.hstack((np.ones((X.shape[0],1)),X))   #prepend ones for intercept

# Plot data distribution
color= ['red' if elem==1 else 'blue' for elem in y ]
plt.scatter(X[:,-2], X[:,-1], c=color)
plt.xlabel('x1')
plt.ylabel('x2')
plt.title('Complete dataset')
# Split into test and training set
X_train=X[:int(X.shape[0]/2)]
X_test=X[int(X.shape[0]/2):]
y_train=y[:int(len(y)/2)]
y_test=y[int(len(y)/2):]
```

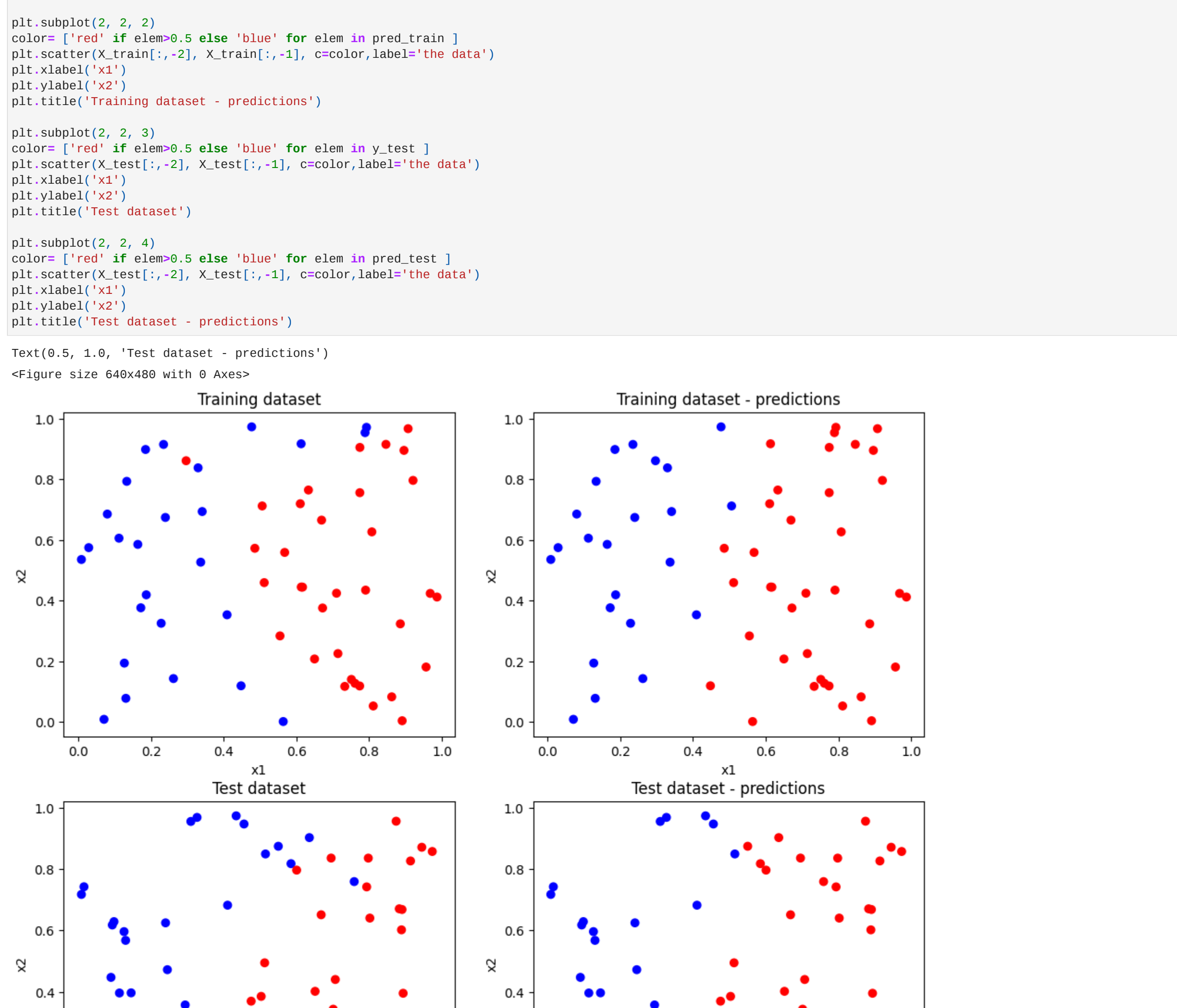


In [11]:
```
#nothing to do here - just execute the cell
w_learned,losses=fitLogRegModel(X_train, y_train)
pred_train=predictLogReg(w_learned, X_train) #as a check
pred_test=predictLogReg(w_learned, X_test)
print("The learnt weights are: w =",w_learned)
```

```
40.41758310647038
35.18968450401735
32.852792570161455
30.973459848270412
29.425116798807117
28.135750521194662
27.05116704051657
26.130189791418267
25.341300081752507
24.660149719107842
24.067740724267335
23.549097805826648
23.092297769604276
22.687755633118993
22.327695373696493
22.00575404127515
21.716682850175026
21.45611936073852
21.220412229989314
21.006485187125772
20.81173054181532
20.633925125746643
20.471163421686963
20.32180397044395
20.184426117166645
20.057794870012373
19.940832170005397
19.83259326258996
The learnt weights are: w = [-2.20623556  6.14043363 -1.33865438]
```

In [12]:
```
# Nothing to do here
# Plot training and test dataset
# Plot predictions for training and test dataset

fig = plt.figure()
fig = plt.figure(figsize = (12,10))
plt.subplot(2, 2, 1)
color= ['red' if elem>0.5 else 'blue' for elem in y_train ]
plt.scatter(X_train[:,-2], X_train[:,-1], c=color,label='the data')
plt.xlabel('x1')
plt.ylabel('x2')
plt.title('Training dataset')

plt.subplot(2, 2, 2)
color= ['red' if elem>0.5 else 'blue' for elem in pred_train ]
plt.scatter(X_train[:,-2], X_train[:,-1], c=color,label='the data')
plt.xlabel('x1')
plt.ylabel('x2')
plt.title('Training dataset - predictions')

plt.subplot(2, 2, 3)
color= ['red' if elem>0.5 else 'blue' for elem in y_test ]
plt.scatter(X_test[:,-2], X_test[:,-1], c=color,label='the data')
plt.xlabel('x1')
plt.ylabel('x2')
plt.title('Test dataset')

plt.subplot(2, 2, 4)
color= ['red' if elem>0.5 else 'blue' for elem in pred_test ]
plt.scatter(X_test[:,-2], X_test[:,-1], c=color,label='the data')
plt.xlabel('x1')
plt.ylabel('x2')
plt.title('Test dataset - predictions')
```

Out[12]:
```
Text(0.5, 1.0, 'Test dataset - predictions')
&lt;Figure size 640x480 with 0 Axes&gt;
```



In the following cell the data set `DataSet_LR_a.csv` is loaded§and split into a training set and a test set (50% each). Now you should:

- Classify samples as class `1` if the Logistic Regression returns values ≥ 0.5 and `0` otherwise. Calculate the entries for a confusion matrix and from these values the Accuracy and Balanced Accuracy in the function `calc_acc(prediction, true_values, threshold)` and apply it on the training and on the test sets.
- Provide ROC curves of the classifiers on the test samples and compute the corresponding AUC. Hint: the functions `roc_curve` and `auc` from `sklearn.metrics` might be useful. Make sure to store the calculated value for the AUC in the variable `rocAUC` - this is important for the unit-test.

Code 1.8 (25 points)

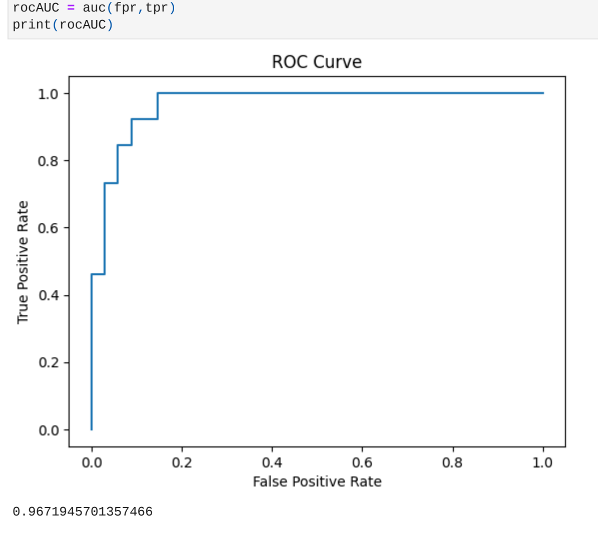
In [13]:
```
"""
Function that calculates the prediction for one or more new samples
@param prediction, np array, predicted values
@param true_values, np array, ground truth

@output pos, float, positive samples
@output neg, float, negative samples
@output tp, float, true positive samples
@output tn, float, true negative samples
@output fp, float, false positive samples
@output fn, float, false negative samples
@output acc, float, accuracy
@output balanced_acc, float, balanced accuracy
"""
def calc_acc(prediction, true_values, threshold = 0.5):
    labels=np.zeros(len(true_values))
    for i in range(len(prediction)):
        if prediction[i]>=threshold:
            labels[i]=1
        elif prediction[i]<threshold:
            labels[i]=0

    logistic_regression_labels=np.zeros(len(true_values))
    for i in range(len(logistic_regression_labels)):
        logistic_regression_labels[i]=labels[i]

    tp,tn,fp,fn=0,0,0,0

    for i in range(len(logistic_regression_labels)):
        if logistic_regression_labels[i]==1 and true_values[i]==1:
            tp+=1
        elif logistic_regression_labels[i]==0 and true_values[i]==0:
            tn+=1
        elif logistic_regression_labels[i]==1 and true_values[i]==0:
            fp+=1
        elif logistic_regression_labels[i]==0 and true_values[i]==1:
            fn+=1

    pos=tp+fn
    neg=tn+fp
    acc=(tp+tn)/(tp+tn+fp+fn)
    tpr=tp/pos
    tnr=tn/neg
    balanced_acc=(tpr+tnr)/2

    return pos, neg, tp, tn, fp, fn, acc, balanced_acc
```

In [14]:
```
# Calculate accuracy and balanced accuracy for test set

result_train = calc_acc(pred_train, y_train)
result_test = calc_acc(pred_test, y_test)
print(result_train[-2])
print(result_train[-1])
print(result_test[-2])
print(result_test[-1])
```

```
0.8833333333333333
0.8744343891402715
0.8666666666666667
0.8823529411764706
```

Code 1.9 (5 points)

In [15]:
```
fpr,tpr,threshold=roc_curve(y_test,pred_test)
plt.title("ROC Curve")
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.plot(fpr,tpr)
plt.show()

rocAUC = auc(fpr,tpr)
print(rocAUC)
```




```
0.9671945701357466
```