Lab 2 Evaluation

In [1]:
```
# Import some useful libraries and functions
import numpy as np
import pandas
def print_stats(dataset):
    """Print statistics of a dataset"""
    print(pandas.DataFrame(dataset).describe())
```

In [2]:
```
# Download the data set and place in the current folder (works on linux only)
filename = 'yacht_hydrodynamics.data'
import os.path
import requests
if not os.path.exists(filename):
    print("Downloading dataset...")
    r = requests.get('https://arbimo.github.io/tp-supervised-learning/tp2/' + filename)
    open(filename , 'wb').write(r.content)
print('Dataset available')
```

```
Dataset available
```

Explore the dataset

- How many examples are there in the dataset? There are 308 examples in the dataset (308 rows)
- How many features for each example? There are 6 features (7 columns), the right-most column being the result of each example
- What is the ground truth of the 10th example? The ground truth of this example is 1.83

In [3]:
```
# loads the dataset and slip between inputs (X) and ground truth (Y)
dataset = np.genfromtxt("yacht_hydrodynamics.data", delimiter='')
X = dataset[:, :-1] # examples features
Y = dataset[:, -1]  # ground truth

# Print the first 5 examples
for i in range(0,5):
    print(f"f({X[i]}) = {Y[i]}")
```

```
f([-5.    0.6   4.78  4.24  3.15  0.35]) = 8.62
f([-5.     0.565  4.77   3.99   3.15   0.15 ]) = 0.18
f([-2.3    0.565  4.78   5.35   2.76   0.15 ]) = 0.29
f([-5.     0.6    4.78   4.24   3.15   0.325]) = 6.2
f([0.    0.53  4.78  3.75  3.15  0.175]) = 0.59
```

The following command adds a column to the inputs.

- What is in the value added this column? The column added only contains ones.
- Why are we doing this? This value represents the "w0" in the formula of the affine function: y = w0 + x1w1 +x2w2 + ... + xnwn. This value needs to be initialized and will be changed afterwards (with the evolution of the algorithm). It avoids having only linear functions, as we want affine functions.

In [4]:
```
X = np.insert(X, 0, np.ones((len(X))), axis= 1)
#print_stats(X)
```

Creating the perceptron

In [5]:
```
w = np.zeros(7)

def h(w, x):
    sum = 0
    for i in range(0,len(x)-1):
        sum+=x[i]*w[i]
    return sum
X = dataset[:, :-1] # examples features
Y = dataset[:, -1]  # ground truth

# print the ground truth and the evaluation of h_w on the first example
print("--First example-- \nground truth:", Y[0], "\nevaluation of h_w:", h(w, X[0]))
```

```
--First example--
ground truth: 8.62
evaluation of h_w: 0.0
```

Loss function

Complete the definiton of the loss function below such that, for a **single** example `x` with ground truth `y`, it returns the L_2 loss of h_w on `x`.

In [6]:
```
def loss(w, x, y):
    return (y-h(w,x))**2 #L2(a,b)=(a-b)^2
```

Empirical loss

Complete the function below to compute the empirical loss of h_w on a **set** of examples X with associated ground truths Y.

In [7]:
```
def emp_loss(w, X, Y):
    sum=0
    for i in range(len(X)):
        sum+=(loss(w,X[i],Y[i]))
    return sum/len(X)
```

Gradient update

Complete the function below so that it computes the dw term (the 'update') based on a set of examples `(X, Y)` the step (`alpha`)

In [8]:
```
def compute_update(w, X, Y, alpha):
    dw = np.zeros(len(w))
    for i in range(0,len(w)-1):
        sum=0
        for j in range(0,len(X)-1):
            sum+=(Y[j]-h(w,X[j]))*X[j][i]
        dw[i]=sum*alpha
    return dw

compute_update(w, X, Y, alpha = 10e-7)
```

Out[8]:
```
array([-0.00754539,  0.00181636,  0.01543604,  0.01266542,  0.01033756,
        0.00130721,  0.        ])
```

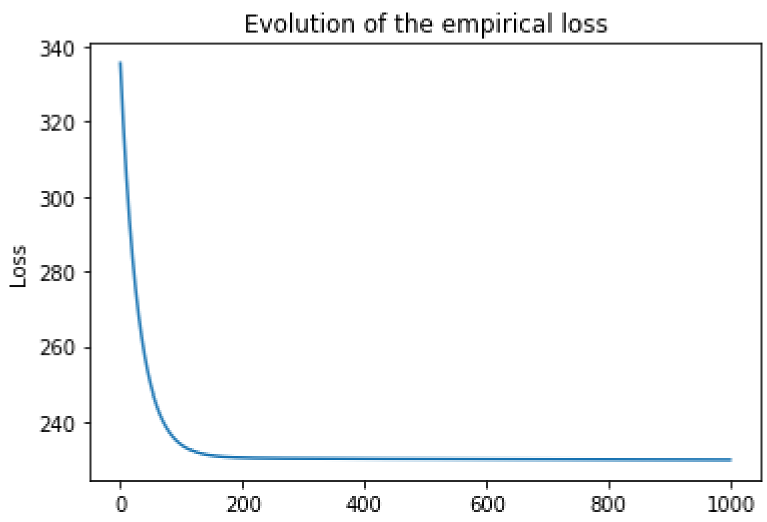
Gradient descent

In [9]:
```
def descent(w_init, X, Y, alpha, max_iter):
    w=np.copy(w_init)
    for i in range(max_iter):
        for j in range (len(w)):
            w[j]=w[j]+compute_update(w, X, Y, alpha)[j]
    return w
```

Exploitation

- Train your perceptron to get a model.
- Visualize the evolution of the loss on the training set. Has it converged? Yes, we can see on the graph below that the loss converged.

In [30]:
```
from matplotlib import pyplot as plt
import statistics

#Same function as descent function but prints the empiral loss for each iteration
def descent_printing_loss(w_init, X, Y, alpha, max_iter):
    l=[]
    w=np.copy(w_init)
    for i in range(max_iter):
        for j in range (len(w)):
            w[j]=w[j]+compute_update(w, X, Y, alpha)[j]
        l.append(emp_loss(w,X,Y))
    plt.title("Evolution of the empirical loss")
    plt.plot(l)
    plt.ylabel('Loss')
    plt.show()
    return w

w_init = np.zeros(7)
descent_printing_loss(w_init, X, Y, 10e-7, 1000)
```



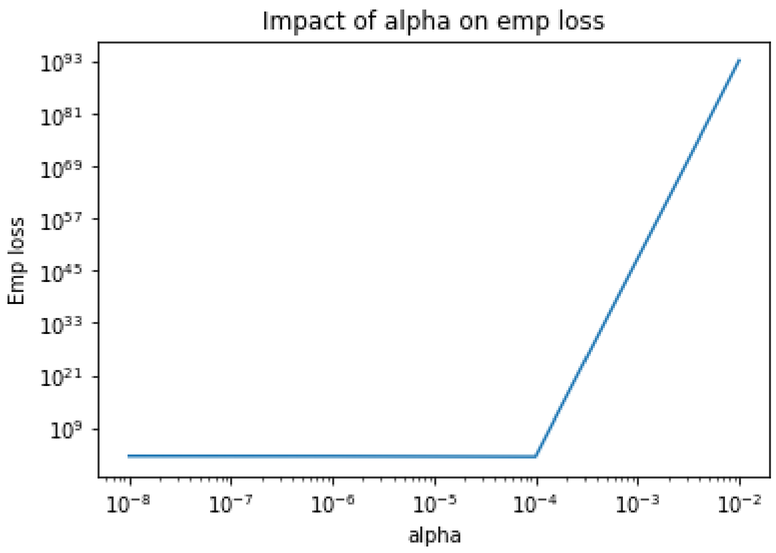
Out[30]:
```
array([-0.18285906,  0.11883795,  1.00261101,  0.74904522,  0.67785455,
        0.4424448 ,  0.        ])
```

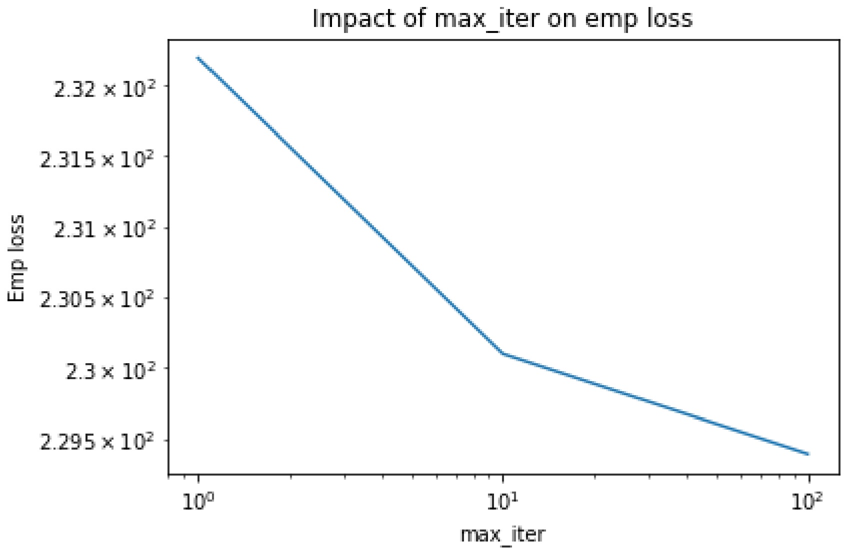
- Try training for several choices of `alpha` and `max_iter` . What seem like a reasonable choice? As we can see on the graphs below, increasing the number of max_iter enables to obtain a better result, but it is time consuming. We decided that max_iter = 100 is a reasonable choice to have a precise model but that does not need too much time to execute. Concerning alpha, we chose 10e-5 to have a compromise between execution time and precision.

In [10]:
```
#Function that constructs plots based on a parameter and the impact of that parameter on weights
from matplotlib import pyplot as plt
def construct_plt(X,Y,parameter,x_axis):
    w_init = np.zeros(7)
    emp_loss_list=[]
    if x_axis=='alpha':
        for i in parameter:
            w = descent(w_init,X,Y,i,10)
            emp_loss_list.append(emp_loss(w,X,Y))
    else:
        for i in parameter:
            w = descent(w_init,X,Y,10e-5,i)
            emp_loss_list.append(emp_loss(w,X,Y))
    plt.xscale('log')
    plt.yscale('log')
    plt.plot(parameter,emp_loss_list)
    plt.xlabel(x_axis)
    plt.ylabel("Emp loss")
    plt.title("Impact of "+x_axis+" on emp loss")
    plt.show()
```

In [11]:
```
def data_analysis(X,Y):
    #Impact of alpha on the weights (the Y axis represents an average on the values in w)
    alphas=[10e-3,10e-5,10e-7,10e-9]
    construct_plt(X,Y,alphas,'alpha')

    #Impact of max_iter on the weights (the Y axis represents an average on the values in w)
    max_iters=[1,10,100]
    construct_plt(X,Y,max_iters,'max_iter')
```

In [12]:
```
%%time
data_analysis(X,Y)
```



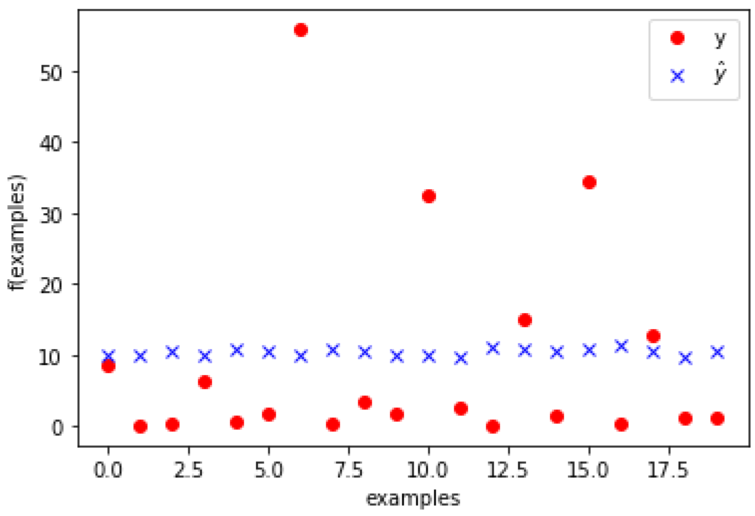



```
Wall time: 5.42 s
```

- What is the loss associated with the final model? The loss associated with our final model is 230.
- Is the final model the optimal one for a perceptron? This final model is not optimal as the loss associated with it is not 0. It means that not all our predictions are accurate; our model is not perfect.

In [13]:
```
# Code sample that can be used to visualize the difference between the ground truth and the prediction
import matplotlib.pyplot as plt

num_samples_to_plot = 20

w_init = np.zeros(7)
w_init = descent(w_init, X, Y, 10e-5, 100)
yw = [h(w_init,x) for x in X]

plt.plot(Y[0:num_samples_to_plot], 'ro', label='y')
plt.plot(yw[0:num_samples_to_plot], 'bx', label='$\hat{y}$')

plt.legend()
plt.xlabel("examples")
plt.ylabel("f(examples)")

print("Loss associated: ", emp_loss(w_init, X, Y))
```

```
Loss associated:  229.39723048470043
```

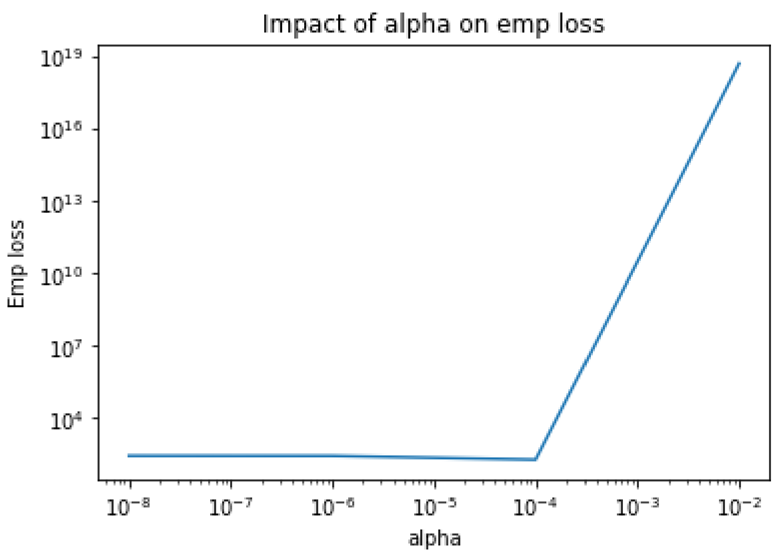


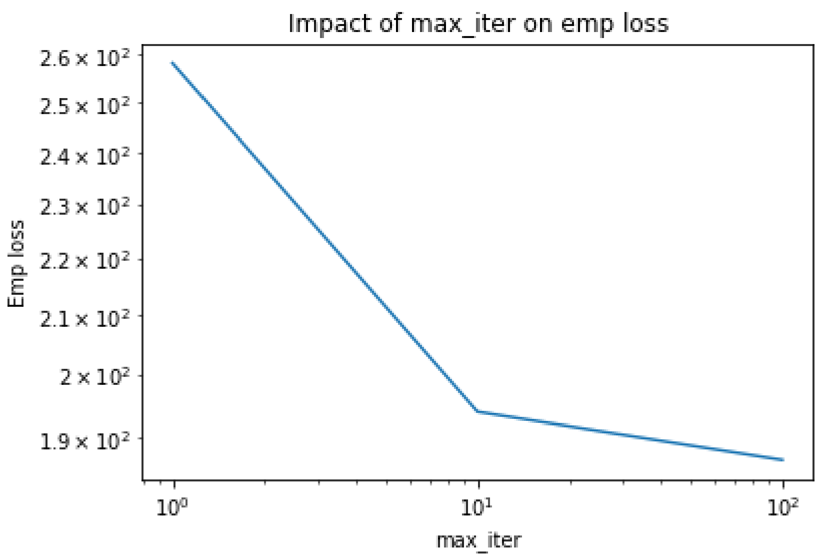
Going further

The Stochastic Gradient Descent algorithm is an improvement over the basic Gradient Descent algorithm in terms of computation time. Indeed, by selecting only a subset of the examples studied, the calculations could be significantly reduced. The subset is selected randomly so it is supposed to represent the dataset well. ***Note: Stochastic means Random***

There are two simple ways to proceed. Either we select random points of the dataset every time we calculate the descent (each time the w vector is updated) with less iterations than the original algorithm, or we could make a smaller randomized copy of the dataset and work with that instead. The second solution being the simplet, that's how we're going to proceed

In [14]:
```
%%time

#We have 308 examples in X. Let's make it 25 random examples
import random
alphas=[10,10e-3,10e-5,10e-7,10e-9]
max_iters=[1,10,100]
X = dataset[:, :-1] # examples features
Y = dataset[:, -1]  # ground truth
randX=[]
randY=[]

for i in range (0,25):
    rand=random.choice(dataset)
    randX.append(rand[:-1])
    randY.append(rand[-1])
#Analysis
data_analysis(randX,randY)
```



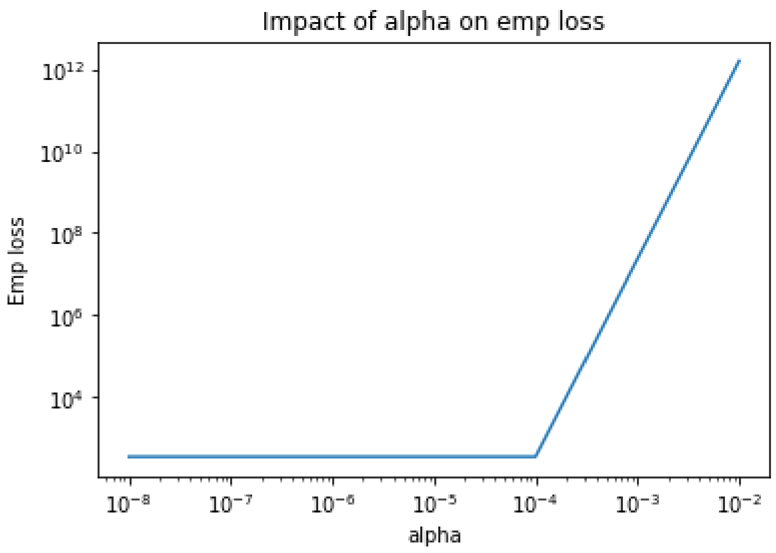


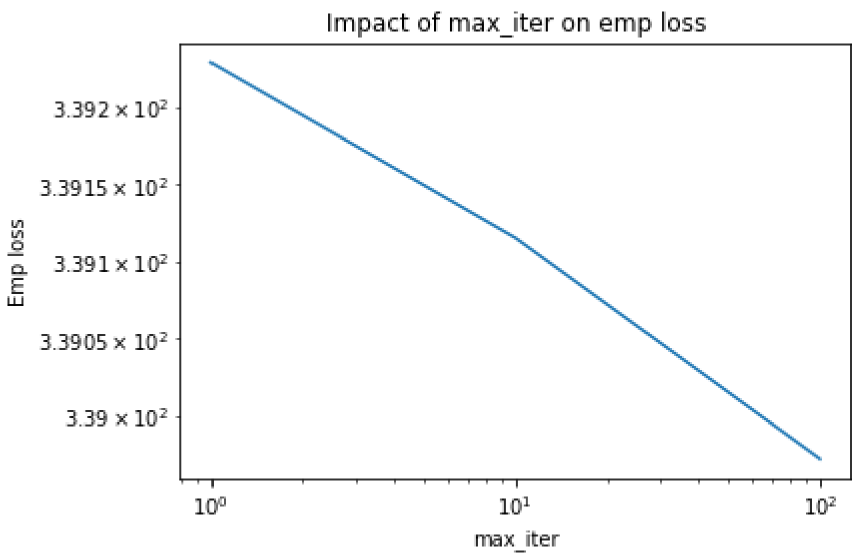

```
Wall time: 1.02 s
```

The plots seem to be fairly similar. Also, we can see that we went from an execution time of 29.1 s to 3.56 s

Data normalization

In [15]:
```
%%time

from sklearn.preprocessing import StandardScaler
sc = StandardScaler(copy=True)
X_normalized = sc.fit_transform(X)
#Analysis
data_analysis(X_normalized,Y)
```






```
Wall time: 6.04 s
```

By comparing the execution time of the two cells, the Data Normalization on X seems to have no impact on the speed on the convergence

In [ ]: