02_perceptron_new

August 15, 2025

1 Machine Learning LAB 2: Perceptrons

Course 2024/25: *F. Chiariotti*

The notebook contains a simple learning task over which we will implement **PERCEPTRON**.

Complete all the **required code sections** and **answer all the questions**.

1.0.1 IMPORTANT for the exam:

The functions you might be required to implement in the exam will have the same signature and parameters as the ones in the labs

1.1 Classification of Stayed/Churned Customers

The Customer Churn table contains information on all 3,758 customers from a Telecommunications company in California in Q2 2022. Companies are naturally interested in churn, i.e., in which users are likely to switch to another company soon to get a better deal, and which are more loyal customers.

The dataset contains three features: - **Tenure in Months**: Number of months the customer has stayed with the company - **Monthly Charge**: The amount charged to the customer monthly - **Age**: Customer's age

The aim of the task is to predict if a customer will churn or not based on the three features.

1.2 Import all the necessary Python libraries and load the dataset

1.2.1 The Dataset

The dataset is a `.csv` file containing three input features and a label. Here is an example of the first 4 rows of the dataset:

Tenure in Months	Monthly Charge	Age	Customer Status
9	65.6	37	0
9	-4.0	46	0
4	73.9	50	1
...	...	...	...

Customer Status is 0 if the customer has stayed with the company and 1 if the customer has churned.

[1]:
```
import numpy as np
import random as rnd
import pandas as pd
from matplotlib import pyplot as plt
from sklearn import linear_model, preprocessing
from sklearn.model_selection import train_test_split

np.random.seed(1) ## Careful! professor used THIS random seed in the
                  ## proposed solution. Use the same otherwise numerical
                  ## results wont match exactly

def load_dataset(filename):
    data_train = pd.read_csv(filename)
    #permute the data
    data_train = data_train.sample(frac=1).reset_index(drop=True) # shuffle the␣
 ↪data
    X = data_train.iloc[:, 0:3].values # Get first two columns as the input
    Y = data_train.iloc[:, 3].values # Get the third column as the label
    Y = 2*Y-1 # Make sure labels are -1 or 1 (0 --> -1, 1 --> 1)
    return X,Y

# Load the dataset
X, Y = load_dataset('data/telecom_customer_churn_cleaned.csv')
```

We are going to differentiate (classify) between **class "1" (churned)** and **class "-1" (stayed)**

1.3 Divide the data into training and test sets

[2]:
```
# Compute the splits
m_training = int(0.75*X.shape[0])

# m_test is the number of samples in the test set (total-training)
m_test =  X.shape[0] - m_training
X_training =  X[:m_training]
Y_training =  Y[:m_training]
X_test =   X[m_training:]
Y_test =  Y[m_training:]

print("Number of samples in the train set:", X_training.shape[0])
print("Number of samples in the test set:", X_test.shape[0])
print("Number of churned users in test:", np.sum(Y_test==-1))
print("Number of loyal users in test:", np.sum(Y_test==1))

# Standardize the input matrix
```

```python
# The transformation is computed on training data and then used on all the 3␣
 ↪sets
scaler = preprocessing.StandardScaler().fit(X_training)

np.set_printoptions(suppress=True) # sets to zero floating point numbers <␣
 ↪min_float_eps
X_training =  scaler.transform(X_training)
print ("Mean of the training input data:", X_training.mean(axis=0))
print ("Std of the training input data:",X_training.std(axis=0))

X_test =  scaler.transform(X_test)
print ("Mean of the test input data:", X_test.mean(axis=0))
print ("Std of the test input data:", X_test.std(axis=0))
```

```
Number of samples in the train set: 2817
Number of samples in the test set: 940
Number of churned users in test: 479
Number of loyal users in test: 461
Mean of the training input data: [-0.  0. -0.]
Std of the training input data: [1. 1. 1.]
Mean of the test input data: [0.0575483  0.05550169 0.0073833 ]
Std of the test input data: [0.98593187 0.97629659 1.00427583]
```

We will use **homogeneous coordinates** to describe all the coefficients of the model.

Hint: The conversion can be performed with the function *hstack* in *numpy*.

[3]:
```python
def to_homogeneous(X_training, X_test):
    # TODO: Transform the input into homogeneous coordinates
    Xh_training = np.hstack((np.ones(shape = (X_training.shape[0], 1)),␣
 ↪X_training))
    Xh_test = np.hstack((np.ones(shape = (X_test.shape[0], 1)), X_test))
    return Xh_training, Xh_test
```

[4]:
```python
# convert to homogeneous coordinates using the function above
X_training, X_test = to_homogeneous(X_training, X_test)
print("Training set in homogeneous coordinates:")
print(X_training[:10])
```

```
Training set in homogeneous coordinates:
[[ 1.         -0.3798618  -1.57020044  0.85174963]
 [ 1.         -0.87925308  0.47180292  1.08667766]
 [ 1.         -0.75440526 -0.6130632  -0.26415851]
 [ 1.         -1.12894873  0.09856916 -0.96894261]
 [ 1.         -1.12894873 -0.58486332 -1.20387064]
 [ 1.          1.78416712  1.39908145  0.08823353]
 [ 1.         -0.7960212  -1.0990965  -0.32289052]
 [ 1.          0.20276137 -0.39907585 -0.96894261]
 [ 1.         -0.62955744  0.63934341  0.96921364]
```

```
 [ 1.         -0.87925308  1.13201197 -0.02923048]]
```

1.4 Deterministic perceptron

Now **complete** the function *perceptron*. The **perceptron** algorithm **does not terminate** if the **data** is not **linearly separable**, therefore your implementation should **terminate** if it **reached the termination** condition seen in class **or** if a **maximum number of iterations** have already been run, where one **iteration** corresponds to **one update of the perceptron weights**. In case the **termination** is reached **because** the **maximum** number of **iterations** have been completed, the implementation should **return the best model** seen throughout.

The current version of the perceptron is **deterministic**: we use a fixed rule to decide which sample should be considered (e.g., the one with the lowest index).

The input parameters to pass are: - X: the matrix of input features, one row for each sample - Y: the vector of labels for the input features matrix X - *max_num_iterations*: the maximum number of iterations for running the perceptron

The output values are: - *best_w*: the vector with the coefficients of the best model (or the latest, if the termination condition is reached) - *best_error*: the *fraction* of misclassified samples for the best model

[5]:
```python
def count_errors(current_w, X, Y):
    # This function:
    # -computes the number of misclassified samples
    # -returns the indexes of all misclassified samples
    # -if there are no misclassified samples, returns -1 as index
    # TODO: write the function
    current_labels = np.sign(np.dot(X, current_w))
    missclass_mask = np.where(current_labels * Y <= 0, True, False)
    n = np.sum(missclass_mask)
    index = -1
    if n > 0:
        index = np.where(missclass_mask == True)[0]
    return n, index


def perceptron_fixed_update(current_w, X, Y):
    # TODO: write the perceptron update function
    n, index = count_errors(current_w, X, Y)
    if (n == 0):
        new_w = current_w
    else:
        x, y = X[index[0], :], Y[index[0]]
        new_w = current_w + x * y
    return new_w

def perceptron_no_randomization(X, Y, max_num_iterations):
    # TODO: write the perceptron main loop
```

```
    # The perceptron should run for up to max_num_iterations, or stop if it␣
↪finds a solution with ERM=0
    #best_error = 10e9
    n_iter = 0
    w = np.zeros(X.shape[1])
    n, _ = count_errors(w, X, Y)
    best_error = n/X.shape[0]
    best_w = w
    while (best_error > 0 and n_iter < max_num_iterations):
        n_iter += 1
        w = perceptron_fixed_update(w, X, Y)
        n , _ = count_errors(w, X, Y)
        error = n/X.shape[0]
        #print("Current error: " + str(n))
        if  error < best_error:
            best_w = w
            best_error = error
    return best_w, best_error
```

Now we use the implementation above of the perceptron to learn a model from the training data using 30 iterations and print the error of the best model we have found.

[6]:
```
w_found, error = perceptron_no_randomization(X_training,Y_training, 30)
print("Training Error of perceptron (30 iterations): " + str(error))
w_found2, error2 = perceptron_no_randomization(X_training,Y_training, 100)
print("Training Error of perceptron (100 iterations): " + str(error2))
```

```
Training Error of perceptron (30 iterations): 0.2751153709620163
Training Error of perceptron (100 iterations): 0.2751153709620163
```

Now use the best model *w_found* to **predict the labels for the test dataset** and print the fraction of misclassified samples in the test set (the test error that is an estimate of the true loss).

[7]:
```
def loss_estimate(w,X,Y):
    # TODO Estimate the test loss
    n, _ = count_errors(w, X, Y)
    t_loss_estimate = n / X.shape[0]
    return t_loss_estimate



true_loss_estimate =  loss_estimate(w_found, X_test, Y_test)       # Error rate␣
↪on the test set
true_loss_estimate2 =  loss_estimate(w_found2, X_test, Y_test)

print("Test Error of perceptron (30 iterations): " + str(true_loss_estimate))
print("Test Error of perceptron (100 iterations): " + str(true_loss_estimate2))
```

```
Test Error of perceptron (30 iterations): 0.26382978723404255
Test Error of perceptron (100 iterations): 0.26382978723404255
```

1.4.1 Randomized perceptron

Implement the correct randomized version of the perceptron such that at each iteration the algorithm picks a random misclassified sample and updates the weights using that sample. The functions will be very similar, except for some minor details.

[9]:
```
def perceptron_randomized_update(current_w, X, Y):
    # TODO: write the perceptron update function
    current_labels = np.sign(np.dot(X, current_w))
    missclass_mask = np.where(current_labels * Y <= 0, True, False)
    if np.any(missclass_mask) > 0:
        index = np.random.choice(a = np.where(missclass_mask == True)[0], size␣
 ↪= 1)[0]
        new_w = current_w + X[index, :] * Y[index]
    else:
        new_w = current_w
    return new_w

def perceptron_with_randomization(X, Y, max_num_iterations):
    # TODO: write the perceptron main loop
    # The perceptron should run for up to max_num_iterations, or stop if it␣
 ↪finds a solution with ERM=0
    n_iter = 0
    w = np.zeros(X.shape[1])
    n, _ = count_errors(w, X, Y)
    best_error = n/X.shape[0]
    best_w = w
    while (best_error > 0 and n_iter < max_num_iterations):
        n_iter += 1
        w = perceptron_randomized_update(w, X, Y)
        n , _ = count_errors(w, X, Y)
        error = n/X.shape[0]
        if error < best_error:
            best_w = w
            best_error = error
    return best_w, best_error
```

Now test the correct version of the perceptron using 30 iterations and print the error of the best model we have found.

[10]:
```
# Now run the perceptron for 30 iterations
w_found, error = perceptron_with_randomization(X_training, Y_training, 30)
w_found2, error2 = perceptron_with_randomization(X_training, Y_training, 100)
print("Training Error of perceptron (30 iterations): " + str(error))
print("Training Error of perceptron (100 iterations): " + str(error2))

true_loss_estimate =  loss_estimate(w_found, X_test, Y_test)       # Error rate␣
 ↪on the test set
```

```python
true_loss_estimate2 =  loss_estimate(w_found2, X_test, Y_test)

print("Test Error of perceptron (30 iterations): " + str(true_loss_estimate))
print("Test Error of perceptron (100 iterations): " + str(true_loss_estimate2))
```

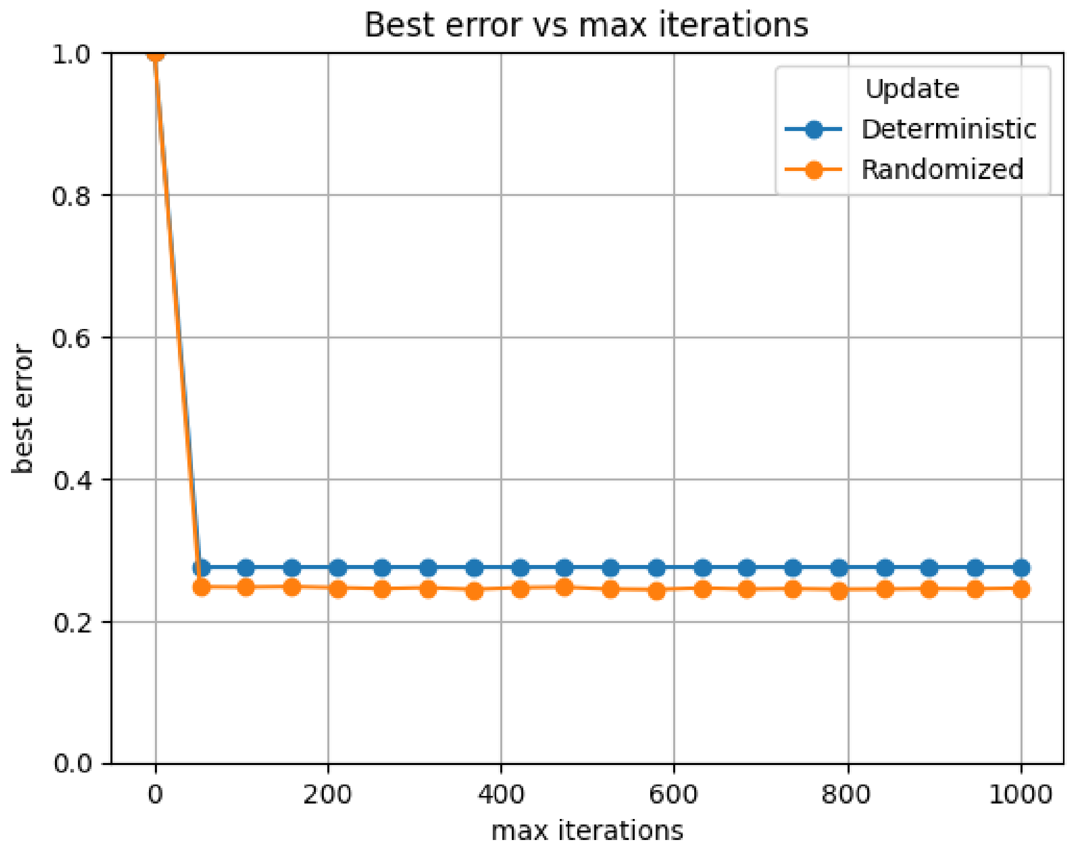
```
Training Error of perceptron (30 iterations): 0.25097621583244584
Training Error of perceptron (100 iterations): 0.24565140220092296
Test Error of perceptron (30 iterations): 0.251063829787234
Test Error of perceptron (100 iterations): 0.2478723404255319
```

[11]:
```python
# TODO Plot the loss with respect to the number of iterations (up to 1000)

errors_no_randomization = []
errors_with_randomization = []

n_iterations = np.linspace(0, 1000, 20)
for n_iter in n_iterations:
    _, error = perceptron_no_randomization(X_training, Y_training, n_iter)
    errors_no_randomization.append(error)
    _, error = perceptron_with_randomization(X_training, Y_training, n_iter)
    errors_with_randomization.append(error)
```

[12]:
```python
fig, ax = plt.subplots()
ax.plot(n_iterations, errors_no_randomization, marker = "o", label =␣
 ↪"Deterministic")
ax.plot(n_iterations, errors_with_randomization, marker = "o", label =␣
 ↪"Randomized")
ax.legend(title = "Update")
ax.set_title("Best error vs max iterations")
ax.set_xlabel("max iterations")
ax.set_ylabel("best error")
plt.ylim((0., 1.))
plt.grid()
plt.show()
```


Best error vs max iterations
Update
Deterministic
Randomized
best error
max iterations
1.0
0.8
0.6
0.4
0.2
0.0
0
200
400
600
800
1000