02_perceptron

February 1, 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(42)

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 three 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
# 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: 465
Number of loyal users in test: 475
Mean of the training input data: [ 0. -0. -0.]
Std of the training input data: [1. 1. 1.]
Mean of the test input data: [ 0.0134851   0.04850383 -0.0433016 ]
Std of the test input data: [1.00014294 1.00683022 1.02078989]
```

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]:
```
def to_homogeneous(X_training: np.ndarray, X_test: np.ndarray) -> tuple[np.
 ↪ndarray, np.ndarray]:
    # TODO: Transform the input into homogeneous coordinates
    nrows, _  = X_training.shape
    Xh_training = np.hstack( (np.ones(shape= (nrows, 1) ), X_training) )
    nrows, _  = X_test.shape
    Xh_test = np.hstack( (np.ones(shape= (nrows, 1) ), X_test) )
    return Xh_training, Xh_test
```

[4]:
```
# 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.          1.2361321   0.87798477 -0.16001986]
 [ 1.          0.10884685  0.4417593   1.37363294]
 [ 1.          1.69539647 -1.57223186 -0.04204657]
 [ 1.          0.15059816 -0.93544295  0.84275312]
 [ 1.          0.56811122 -0.38890759 -0.57292638]
 [ 1.         -0.39216881 -1.41010975 -0.39596645]
 [ 1.         -1.0184384  -1.53880462 -1.04481955]
 [ 1.         -0.35041751 -0.71649454  0.72477983]
 [ 1.         -1.18544362 -1.45857925  0.4298466 ]
 [ 1.          1.44488863 -1.4385229  -0.16001986]]
```

1.4 Deterministic perceptron (Pocket Algorithm)

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: np.ndarray, X:np.ndarray, Y: np.ndarray):
    # 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

    """ This works but is SLOW
    n = 0
    indexes = []
    for i in range( X.shape[0] ):
        if ( np.dot(X[i, :], current_w) * Y[i] ) <= 0:
            n+=1
            indexes.append(i)
    if not indexes:
        indexes = [-1]
    return n, indexes
    """

    # Use numpy built-in methods for better performance!
    # Time gain is huge !! T_new_method = 1/30 * T_old_method
    indexes = np.nonzero(np.sign(np.dot(X, current_w)) - Y)[0] #misclassified
    n = len(indexes)
    if n== 0:
        indexes = [-1]
    return n, indexes
```

```python
def perceptron_fixed_update(current_w: np.ndarray, X:  np.ndarray, Y:  np.
 ↪ndarray):
    # TODO: write the perceptron update function
    n, indexes = count_errors(current_w, X, Y)
    if n == 0:
        new_w = current_w
    else:
        new_w = current_w + Y[indexes[0]] *  X[indexes[0], :]
    return new_w

def perceptron_no_randomization(X:  np.ndarray, Y:  np.ndarray,␣
 ↪max_num_iterations: int):
    # 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

    """
    # Random inizialization
    idx = np.random.choice(np.arange(0, X.shape[0]), size = 1)
    w = X[idx[0],:] * Y[idx[0]]
    n, _ = count_errors(w, X, Y)
    error = n / X.shape[0]
    """
    # Zero weight inizialization
    w = np.zeros(shape= X.shape[1])
    n, _ = count_errors(w, X, Y)
    error = n / X.shape[0]


    best_error = error
    """This is WRONG !!!
    best_w = w
    """
    best_w = w.copy()

    iters = 0
    for _ in range(max_num_iterations):
        iters += 1
        w = perceptron_fixed_update(w, X, Y)
        n, _ = count_errors(w, X, Y)
        error = n / X.shape[0]

        if error <  best_error:
            best_error = error
            best_w = w.copy() # Be careful!!
        if  error == 0:
            break

    return best_w, best_error, iters
```

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, iters = perceptron_no_randomization(X_training,Y_training, 30)
print("Training Error of perceptron (30 iterations): " + str(error) + ".␣
 ↪Effective iterations: " + str(iters))
w_found2, error2, iters = perceptron_no_randomization(X_training,Y_training,␣
 ↪100)
print("Training Error of perceptron (100 iterations): " + str(error2) + ".␣
 ↪Effective iterations: " + str(iters))
```

```
Training Error of perceptron (30 iterations): 0.24565140220092296. Effective
iterations: 30
Training Error of perceptron (100 iterations): 0.24565140220092296. Effective
iterations: 100
```

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): 0.25
Test Error of perceptron (100 iterations): 0.25
```

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.

[8]:
```
def perceptron_randomized_update(current_w, X, Y):
    # TODO: write the perceptron update function
    n, indexes = count_errors(current_w, X, Y)
    if n == 0:
        new_w = current_w.copy()
    else:
        idx = np.random.choice(indexes, size = 1)
        new_w = current_w + Y[idx[0]] *  X[idx[0], :]
    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

    """
    # Random initialization
    idx = np.random.choice(np.arange(0, X.shape[0]), size = 1)
    w = X[idx[0],:] * Y[idx[0]]
    n, _ = count_errors(w, X, Y)
    error = n / X.shape[0]
    """
    # Zero weight inizialization
    w = np.zeros(shape= X.shape[1])
    n, _ = count_errors(w, X, Y)
    error = n / X.shape[0]

    best_error = error
    best_w = w.copy()

    iters = 0
    for _ in range(max_num_iterations):
        iters += 1
        # Random update
        w = perceptron_randomized_update(w, X, Y)
        n, _ = count_errors(w, X, Y)
        error = n / X.shape[0]

        if error <  best_error:
            best_error = error
            best_w = w.copy()

        if  error == 0:
            break

    return best_w, best_error, iters
```

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

[9]:
```
# Now run the perceptron for 30 iterations
w_found, error, iters = perceptron_with_randomization(X_training, Y_training,␣
 ↪30)
w_found2, error2, iters2 = perceptron_with_randomization(X_training,␣
 ↪Y_training, 100)
print("Training Error of perceptron (30 iterations): " + str(error) + ".␣
 ↪Effective iterations: " + str(iters))
print("Training Error of perceptron (100 iterations): " + str(error2) + ".␣
 ↪Effective iterations: " + str(iters2))

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))
```

```
Training Error of perceptron (30 iterations): 0.2552360667376642. Effective
iterations: 30
Training Error of perceptron (100 iterations): 0.24671636492722754. Effective
iterations: 100
Test Error of perceptron (30 iterations): 0.25851063829787235
Test Error of perceptron (100 iterations): 0.251063829787234
```

[10]:
```
# TODO Plot the loss with respect to the number of iterations (up to 1000)
iters_array = np.arange(30, 1000, 20)
errors_fixed_update = []
errors_randomized_update = []
for max_iters in iters_array:
    print(max_iters)
    w, error, _ = perceptron_no_randomization(X_training, Y_training, max_iters)
    errors_fixed_update.append( error )
    w, error, _ = perceptron_with_randomization(X_training, Y_training,␣
 ↪max_iters)
    errors_randomized_update.append( error )
```

```
30
50
70
90
110
130
150
170
190
210
230
250
```

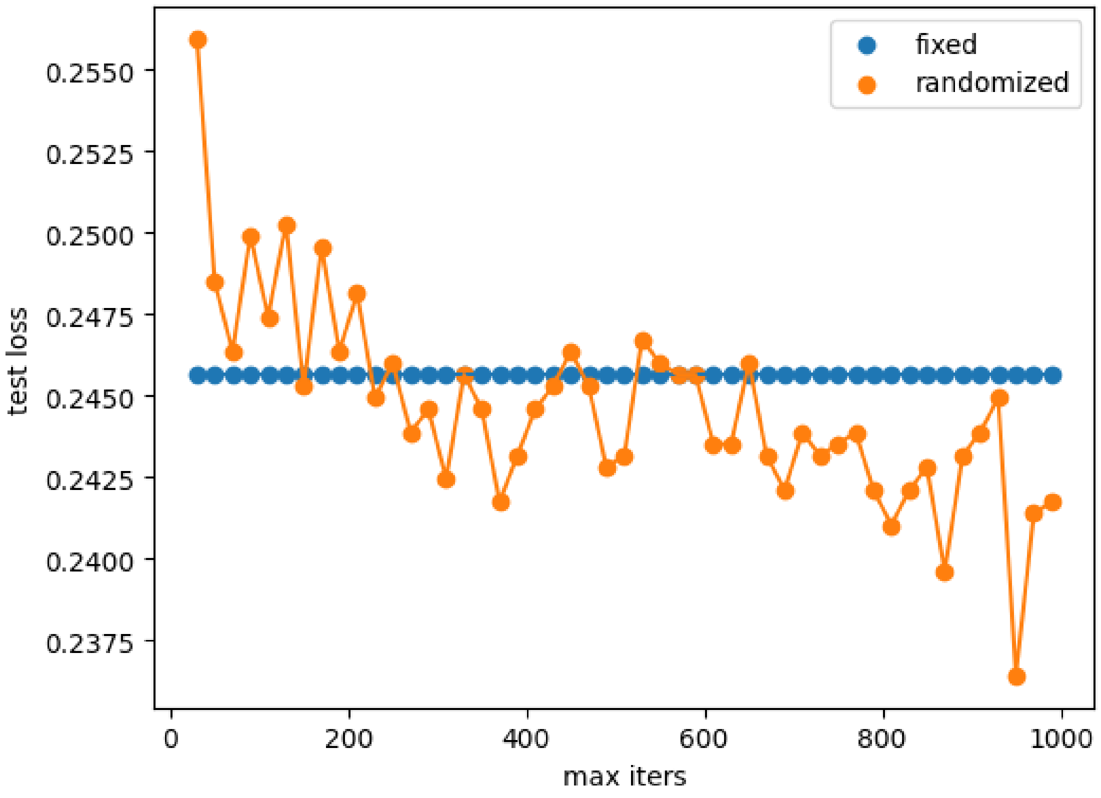
```
270
290
310
330
350
370
390
410
430
450
470
490
510
530
550
570
590
610
630
650
670
690
710
730
750
770
790
810
830
850
870
890
910
930
950
970
990
```

[11]:
```
import matplotlib.pyplot as plt
fig, ax = plt.subplots()
ax.plot(iters_array, errors_fixed_update)
ax.scatter(iters_array, errors_fixed_update, label = "fixed")

ax.plot(iters_array, errors_randomized_update)
ax.scatter(iters_array, errors_randomized_update, label = "randomized")

ax.set_xlabel("max iters")
ax.set_ylabel("test loss")
ax.legend(loc = "best")
```

[11]:
```
<matplotlib.legend.Legend at 0x141f761e0>
```



[12]:
```
a = np.array([1,2,3,4])
b = a
b[0] = 0
print(a)
```

```
[0 2 3 4]
```

1.4.2 My final comments:

The results where expected. I initialized the weights as the zero vector for both the deterministic and randomized perceptron.

The deterministic perceptron always chooses the first misclassified sample to update the weights. Since the initial weight is also fixed, the update rule is completely deterministic. The deterministic perceptron with max_iters = 100 and max_iters = 1000 behave identically up to the iteration = 100. Then the latter goes on and can potentially find a smallest error. Therefore *the error of the deterministic perceptron is a monotonically decreasing function of max_iters.*

For the randomized perceptron this is not true. Generally, because of the randomness in the choice of the update, the randomized perceptron explores more possibilities and this results in the fact that it is able to find a smaller error than the deterministic counterpart.

1.4.3 Careful: BIG mistake to avoid!!

Always check wheter you are working with *views* or *copies* !!!

```
a = np.array([1,2,3,4])
b = a
b[0] = 0
print(a) # [0, 2, 3, 4]
```

this caused a big error in the computation of best_w: since best_w was assigned as a *view* of current_w, updating best_w also resulted in the modification of current_w !!!

1.5 Initialization: setting zero weights is important!

Indeed $w_{initial} = \vec{0}$ is an assumption of the theorem that guarantees perceptron convergence in the separable case.