perceptron_learning_stud

September 22, 2019

0.1 Perceptron Learning Rule

Last revision: Martin Melchior - 18.09.2019

0.1.1 Preparation of the Data

1 STUDENT: Sebastian Häni haeniseb@students.zhaw.ch

Instead of providing a fixed input dataset, we here generate it randomly. For easier comparison, we want to make sure that the same data is produced. Therefore, we set a random seed (set to 1 below).

Furthermore, we provide a suitable plotting utility that allows you to inspect the generated data.

[1]:
```
import numpy as np

def prepare_data(m,m1,a,s,width=0.6,eps=0.5, seed=1):
    """
    Generates a random linearly separable 2D test set and associated labels␣
 ↪(0|1).
    The x-values are distributed in the interval [-0.5,0.5].
    With the parameters a,s you can control the line that separates the two␣
 ↪classes.
    This turns out to be the line with the widest corridor between the two␣
 ↪classes (with width 'width').
    If the random seed is set, the set will always look the same for given␣
 ↪input parameters.

    Arguments:
    a -- y-intercept of the seperating line
    s -- slope of the separating line
    m -- number of samples
    m1 -- number of samples labelled with '1'
    width -- width of the corridor between the two classes
    eps -- measure for the variation of the samples in x2-direction

    Returns:
    x -- generated 2D data of shape (2,n)
```

```
    y -- labels (0 or 1) of shape (1,n)
    """
    np.random.seed(seed)
    idx = np.random.choice(m, m1, replace=False)
    y = np.zeros(m, dtype=int).reshape(1,m)
    y[0,idx] = 1

    x = np.random.rand(2,m).reshape(2,m) # random numbers uniformly distributed␣
 ↪in [0,1]
    x[0,:]-= 0.5
    idx1 = y[0,:]==1
    idx2 = y[0,:]==0
    x[1,idx1] = (a+s*x[0,idx1]) + (width/2+eps*x[1,idx1])
    x[1,idx2] = (a+s*x[0,idx2]) - (width/2+eps*x[1,idx2])

    return x,y
```

[2]:
```
import matplotlib.pyplot as plt
%matplotlib inline

def line(a, s, n=100):
    """
    Returns a line 2D array with x and y=a+s*x.

    Parameters:
    a -- intercept
    s -- slope
    n -- number of points

    Returns:
    2d array of shape (n,2)
    """
    x = np.linspace(-0.5, 0.5, n)
    l = np.array([x,a+s*x]).reshape(2,n)
    return l

def plot(x, y, params_best=None, params_before=None, params_after=None,␣
 ↪misclassified=None, selected=None):
    """
    Plot the 2D data provided in form of the x-array.
    Use markers depending on the label ('1 - red cross, 0 - blue cross').
    Optionally, you can pass tuples with parameters for a line (a: y-intercept,␣
 ↪s: slope)
    * params_best: ideal separating line (green dashed)
    * params: predicted line (magenta)
    Finally, you can also mark single points:
    * misclassified: array of misclassified points (blue circles)
```

```
    * selected: array of selected points (green filled circles)

    Parameters:
    x -- 2D input dataset of shape (2,n)
    y -- ground truth labels of shape (1,n)
    params_best -- parameters for the best separating line
    params -- any line parameters
    misclassified -- array of points to be marked as misclassified
    selected -- array of points to be marked as selected
    """
    idx1 = y[0,:]==1
    idx2 = y[0,:]==0
    plt.plot(x[0,idx1], x[1,idx1], 'r+', label="label 1")
    plt.plot(x[0,idx2], x[1,idx2], 'b+', label="label 0")
    if not params_best is None:
        a = params_best[0]
        s = params_best[1]
        l = line(a,s)
        plt.plot(l[0,:], l[1,:], 'g--')
    if not params_before is None:
        a = params_before[0]
        s = params_before[1]
        l = line(a,s)
        plt.plot(l[0,:], l[1,:], 'm--')
    if not params_after is None:
        a = params_after[0]
        s = params_after[1]
        l = line(a,s)
        plt.plot(l[0,:], l[1,:], 'm-')
    if not misclassified is None:
        plt.plot(x[0,misclassified], x[1,misclassified], 'o',␣
↪label="misclassified")
    if not selected is None:
        plt.plot(x[0,selected], x[1,selected], 'oy', label="selected")

    plt.legend()
    plt.show()
```

1.0.1 Parameters for the decision boundary

Here, you should implement a function that translates the weights vector (w_1, w_2) and the bias b into parameters of a straight line ($x_2 = a + s \cdot x_1$)

[3]:
```
def lineparams(weight, bias):
    """
    Translates the weights vector and the bias into line parameters with a␣
↪x2-intercept 'a' and a slope 's'.
```

```
    Parameters:
    weight -- weights vector of shape (1,2)
    bias -- bias (a number)

    Returns:
    a -- x2-intercept
    s -- slope of the line in the (x1,x2)-plane
    """
    ### START YOUR CODE ###
    a = -(bias / weight[0, 1])
    s = -(weight[0, 0] / weight[0, 1])
    ### END YOUR CODE ###
    return a,s
```

1.0.2 Implement the Perceptron Learning Algorithm

by implementing the functions * predict * update * select_datapoint * train
Follow the descriptions of these functions.

[4]:
```
def predict(x,w,b):
    """
    Computes the predicted value for a perceptron (single LTU).

    Parameters:
    x -- input dataset of shape (2,m)
    w -- weights vector of shape (1,2)
    b -- bias (a number)

    Returns:
    y -- prediction of a perceptron (single LTU) of shape (1,m)
    """
    ### START YOUR CODE ###
    y = np.heaviside(np.dot(w, x) + b, 0.)
    ### END YOUR CODE ###
    return y

def update(x,y,w,b,alpha=1.0):
    """
    Performs an update step in accordance with the perceptron learning␣
 ↪algorithm.

    Parameters:
    x -- input data point of shape (2,1)
    y -- true label ('ground truth') for the specified point
    w -- weight vector of shape (1,2)
    b -- bias (a number)
```

```
    Returns:
    w1 -- updated weight vector
    b1 -- updated bias
    """
    ypred = predict(x,w,b)
    ### START YOUR CODE ###
    w1 = w - alpha * (ypred - y) * x
    b1 = b - alpha * (ypred - y)
    ### END YOUR CODE ###

    return w1, b1


def select_datapoint(x, y, w, b):
    """
    Identifies the misclassified data points and selects one of them.
    In case all datapoints are correctly classified None is returned.

    Parameters:
    x -- input dataset of shape (2,m)
    y -- ground truth labels of shape (1,m)
    w -- weights vector of shape (1,2)
    b -- bias (a number)

    Returns:
    x1 -- one of the wrongly classified datapoint (of shape (2,1))
    y1 -- the associated true label
    misclasssified -- array with indices of wrongly classified datapoints or␣
 ↪empty array
    """
    ypred = predict(x,w,b)
    wrong_mask = (ypred != y)[0]
    misclassified = np.where(wrong_mask)[0]
    if len(misclassified)>0:
        x1 = x[:,misclassified[0]]
        y1 = y[0,misclassified[0]]
        return x1, y1, misclassified
    return None, None, []


def train(weight_init, bias_init, x, y, alpha=1.0, debug=False,␣
 ↪params_best=None, max_iter=1000):
    """
    Trains the perceptron (single LTU) for the given data x and ground truth␣
 ↪labels y
```

```
    by using the perceptron learning algorithm with learning rate alpha␣
↪(default is 1.0).
    The max number of iterations is limited to 1000.

    Optionally, debug output can be provided in form of plots with showing the␣
↪effect
    of the update (decision boundary before and after the update) provided at␣
↪each iteration.

    Parameters:
    weight_init -- weights vector of shape (1,2)
    bias_init -- bias (a number)
    x -- input dataset of shape (2,m)
    y -- ground truth labels of shape (1,m)
    alpha -- learning rate
    debug -- flag for whether debug information should be provided for each␣
↪iteration
    params_best -- needed if debug=True for plotting the true decision boundary

    Returns:
    weight -- trained weights
    bias -- trained bias
    misclassified_counts -- array with the number of misclassifications at each␣
↪iteration
    """
    weight = weight_init
    bias = bias_init
    iterations = 0
    misclassified_counts = [] # we track them to show how the system learned in␣
↪the end

    ### START YOUR CODE ###
    while iterations <= max_iter:
        iterations += 1

        params_before = lineparams(weight, bias)
        x1, y1, misclassified = select_datapoint(x, y, weight, bias)
        n_misclassified = len(misclassified)
        misclassified_counts.append(n_misclassified)

        if n_misclassified == 0:
            break

        weight, bias = update(x1, y1, weight, bias, alpha)

        if debug:
```

```
            params_after = lineparams(weight, bias)
            plot(x, y, params_best=params_best, params_before=params_before,␣
↪params_after=params_after,
                 misclassified=misclassified, selected=np.
↪array([misclassified[0]]))
    ### END YOUR CODE ###

    return weight, bias, misclassified_counts
```

Auxiliary Function

[5]:
```
def weights_and_bias(a,s):
    """
    Computes weights vector and bias from line parameters x2-intercept and␣
↪slope.
    """
    w1 = - s
    w2 = 1.0
    weight = np.array([w1,w2]).reshape(1,2)
    bias = - a
    return weight, bias
```

1.0.3 Test Your Implementation

1/ Prepare the dataset by using the prepare_data function defined above and plot it. Use the parameters specified below (a=1, s=2, n=100, n1=50).

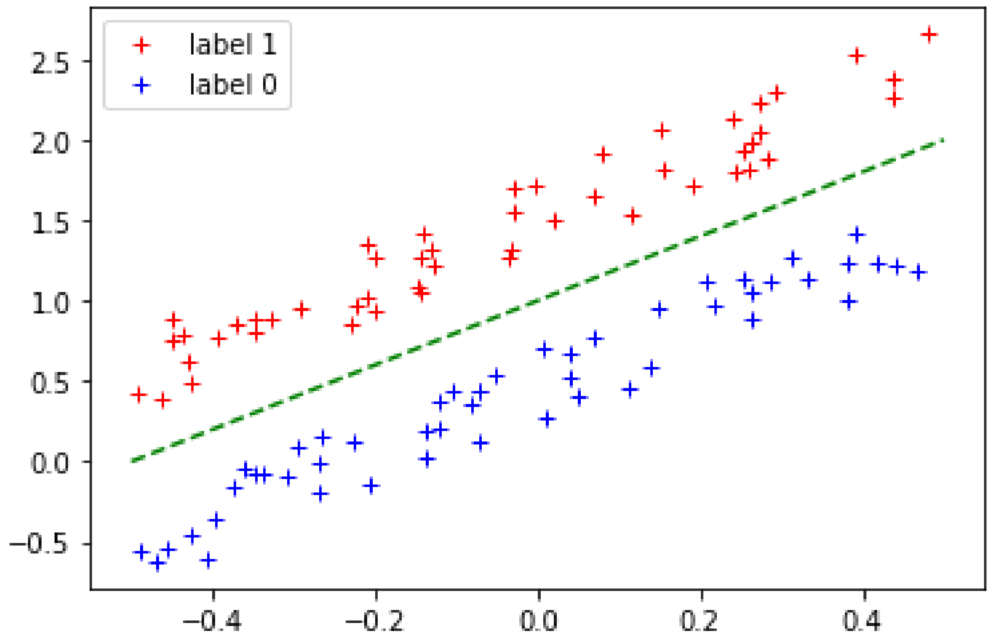
2/ Run the training with the default learning rate (alpha=1). Paste the plots with the situation at the start and with the situation at the end of the training into a text document. Paste also the start parameters (weight and bias) and trained parameters.

3/ Create a plot with the number of mis-classifications vs iteration and paste it into the text document.

1/ Prepare the dataset

[6]:
```
n = 100
n1 = 50
a = 1
s = 2
x,y = prepare_data(n,n1,a,s)

params_best = (a,s)
weight_best, bias_best = weights_and_bias(a, s)
print("weight: ", weight_best, "  bias: ", bias_best)
plot(x,y,params_best=params_best)
```

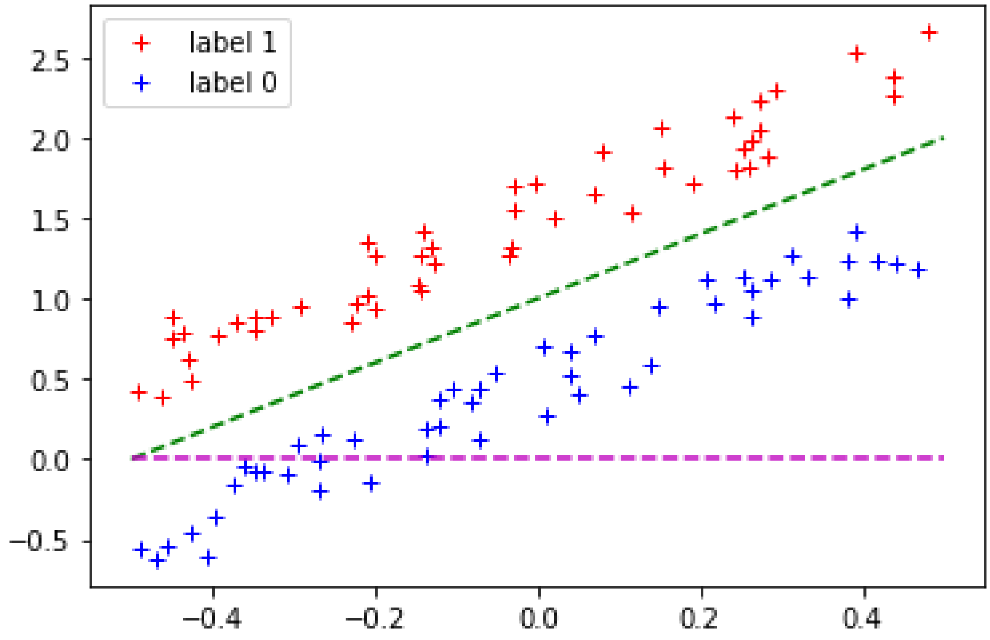
```
weight:  [[-2.  1.]]   bias:  -1
```



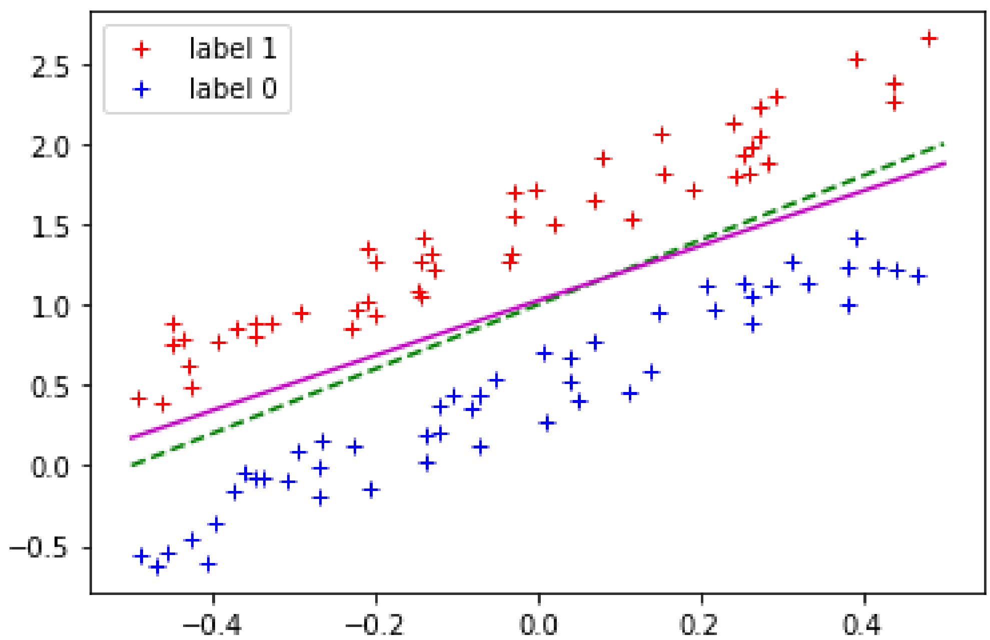
2/ Run the training

[7]:
```
a1 = 0
s1 = 0
alpha = 1.0

weight1, bias1 = weights_and_bias(a1,s1)
print("Initial Params: ",weight1,bias1)
params = lineparams(weight1, bias1)
plot(x,y,params_best, params)

weight1,bias1,misclassified_counts = train(weight1, bias1, x, y, alpha=alpha)
#weight1,bias1,misclassified_counts = train(weight1, bias1, x, y, alpha=alpha,␣
↪debug=True, params_best=params_best)
params = lineparams(weight1, bias1)
print("Iterations: ", len(misclassified_counts)-1)
print("Trained Params: ", weight1,bias1)
plot(x,y, params_best=params_best, params_after=params)
```

```
Initial Params:  [[0. 1.]] 0
```



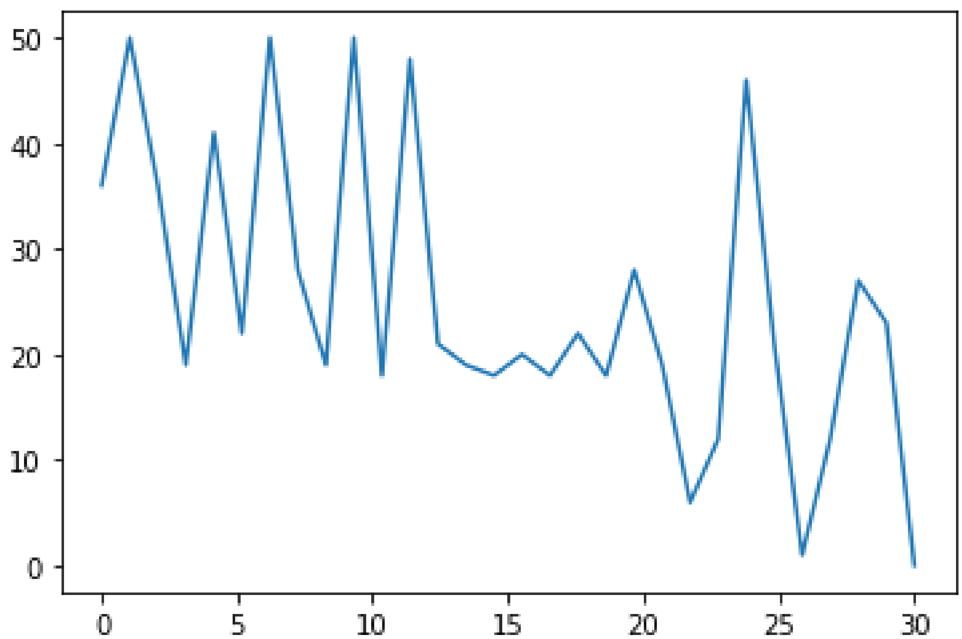

```
Iterations:  29
Trained Params:  [[-4.99046876  2.92867787]] [-3.]
```



3/ Create the plot with the misclassifications per iteration

[8]:
```python
nit = len(misclassified_counts)
it = np.linspace(0,nit,nit)

plt.plot(it, misclassified_counts)
```

[8]:
```
[<matplotlib.lines.Line2D at 0x115515198>]
```



2 Personal findings

Since we used learning rate `alpha=1.0`, the bias was overcorrecting all the time. But the slope was able to overcome the inflexibility through the point data itself.

The jitter in the misclassifications vs iteration plot is due to the high learning rate. If I lower the learning rate, the jumps back up in the misclassification count are occurring much less.

But lowering the learning rate will also likely use up more iterations. So it is always a trade off between performance and optimal solution.

2.1 Student: Sebastian Häni haeniseb@students.zhaw.ch