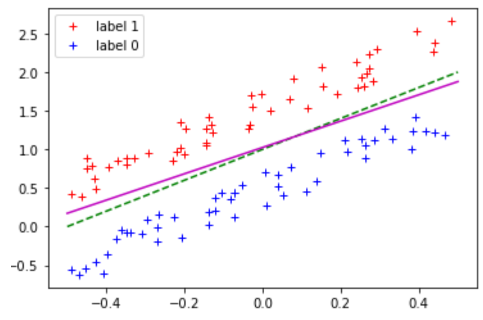
MASTER OF SCIENCE
IN ENGINEERING

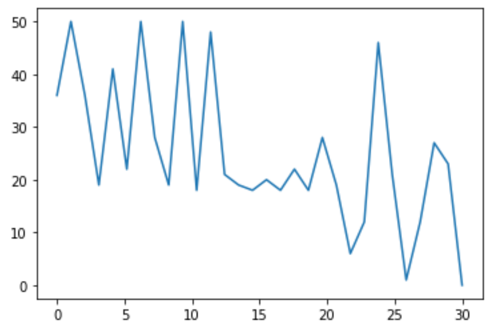
LAB REPORT

Modul	**Deep Learning (TSM_DeLearn)**
Title	**Perceptron Learning Algorithm**
Date	02.03.2021
Objective	• Implement the perceptron learning algorithm. • Test the implementation using: - Dummy dataset which is generated on the fly. - Lightweight MNIST dataset.
Lecturer	• Prof. Dr. Jean Hennebert • Prof. Dr. Martin Melchior
Student	Group 11: • Florian Merz (florian.merz@ost.ch) • Patrick Koller (patrick.koller@ost.ch) • Yohanes Sugiarto (yohanes.sugiarto@ost.ch)

Results step 1 – linearly separable dataset:

A linearly separable 2D dataset, consisting of label 0 (blue crosses) and label 1 (red crosses) is created on the fly. The green, dashed line represents the optimal separation line between the two classes with the maximal margin to each class. Applying the perceptron-learning-algorithm to this dataset results in the magenta colored line, which separates the classes perfectly, but not optimally in a sense to maximize the margin to both classes.

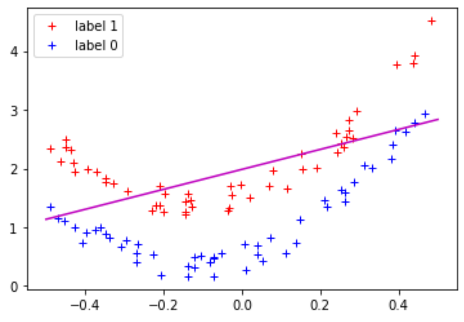


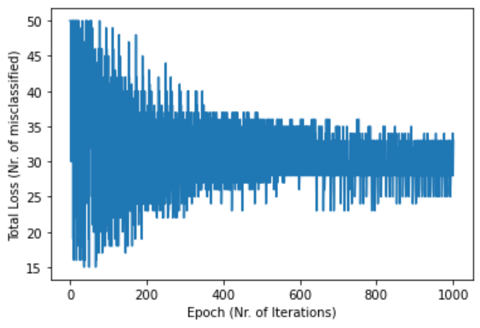


- Linearly separable dataset
  - Optimal parameters (Green dashed line) are w1 = -2.0, w2 = 1.0, and bias = -1
  - Training converged in 29 iterations
  - Resulting parameters are w1 = -4.990, w2 = 2.929 and bias = -3
  - The perceptron-learning-algorithm separates the dataset nicely into the two desired classes

Results step 1 – non-linear separable dataset: (Optional)

A quadratic separable 2D dataset, consisting of label 0 (blue crosses) and label 1 (red crosses) is created on the fly. Applying the perceptron-learning-algorithm to this quadratic dataset results in the linear magenta colored line, which fails to separate the dataset in a meaningful way.

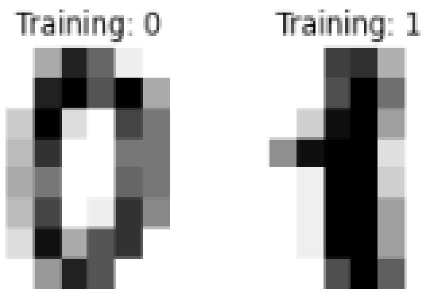


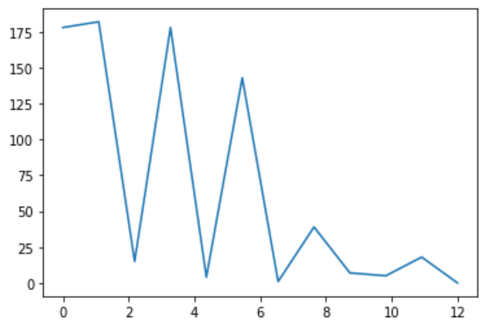


- Quadratic separable dataset
  - The perceptron-learning-algorithm fails to separate the dataset as desired
  - Optimal solution would need to be a quadratic function

Results step 2 – MNIST dataset:

The lightweight MNIST dataset (which consists of 8×8-pixel images of handwritten digits) is widely used for training and testing in the field of machine learning and deep learning. Applying the perceptron-learning-algorithm to differentiate between the numbers 0 and 1 converges after only 11 iterations and gives perfect results in terms of accuraccy.





- MNIST dataset
  - The perceptron-learning-algorithm is able to create a nice decision boundary to separate the numbers 0 and 1 successfully
  - Comparing all other pairs of numbers is much more difficult. Our implementation is not able to differentiate clearly better than random between all other pairs of numbers.

perceptron_learning_merz_mnist

March 2, 2021

0.1 Perceptron Learning Rule

Last revision: Martin Melchior - 18.09.2019

In this exercise you implement the perceptron learning rule. Then you apply it to linearly separable data (the data can be generated on the fly) and you can convince yourself that your system is properly implemented and has found the separating line.

Some emphasis should be given to properly handle numpy arrays. These will be much more extensively used in upcoming exercises of later weeks. So, we recommend to take a serious glance at them.

0.1.1 Preparation of the Data

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).

The data will be generated in form of a 2d array, the first index enumerating the dimensions (rows, in the 2d case index 0 and 1), the second enumerating the samples (columns).

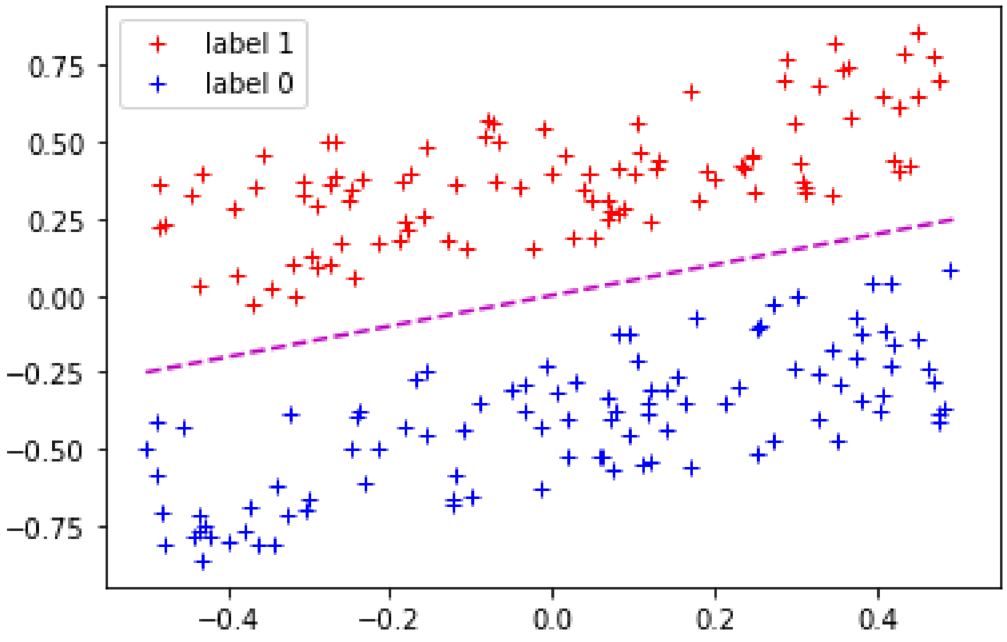
Furthermore, we provide a suitable plotting utility that allows you to inspect the generated data.

[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()
```

Generate and Plot a Sample

```
[3]: x,y = prepare_data(200,100,0,0.5,width=0.3,eps=0.5, seed=1)
     plot(x, y, params_before=(0,0.5))
```



0.1.2 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 ($x_2 = a + s \cdot x_1$)

```
[4]: 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 = -b/w2
         a = -bias/weight[0,1]

         # s = -w1/w2
```

```
    s = -weight[0,0]/weight[0,1]

    ### END YOUR CODE ###
    return a,s
```

0.1.3 Implement the Perceptron Learning Algorithm

by implementing the functions * predict * update * select_datapoint * train

Follow the description of these functions.

[5]:
```
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 ###
    # h = w.x + b -> hi = w0.x0i + w1.x1i + b
    h = np.dot(w, x) + b
    #print("h: ", h)

    # y = 1 if h >= 0
    # y = 0 if h < 0
    y = np.where(h >= 0, 1, 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
    w1 = w - alpha*(ypred - y)*x

    # b1 = b - alpha.(ypred - y)
    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:
        # Identifies the misclassified data points
        x1, y1, misclassified = select_datapoint(x, y, weight, bias)

        # Insert number of misclassifieds into misclassified_counts
        misclassified_counts.append(len(misclassified))

        params_before = lineparams(weight, bias)

        # Any misclassified? No? -> break training
        if len(misclassified) == 0:
            break
        else:
            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]]))

        iterations = iterations + 1

    ### END YOUR CODE ###

    return weight, bias, misclassified_counts
```

[6]:
```
# FUNCTIONS TESTS

# Test predict(x,w,b)->y    ([2xm],[1x2],@)->[1xm]
xt = np.array([[1,2,3], [4,5,-6], [-2,-3,-5]])
print(xt.shape)

wt = np.array([[1,2,1]])
print(wt.shape)

bt = 5

yt = predict(xt,wt,bt)
print(yt.shape)
print(yt)
```

```
(3, 3)
(1, 3)
(1, 3)
[[1 1 0]]
```

Auxiliary Function

[7]:
```
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
```

0.1.4 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).

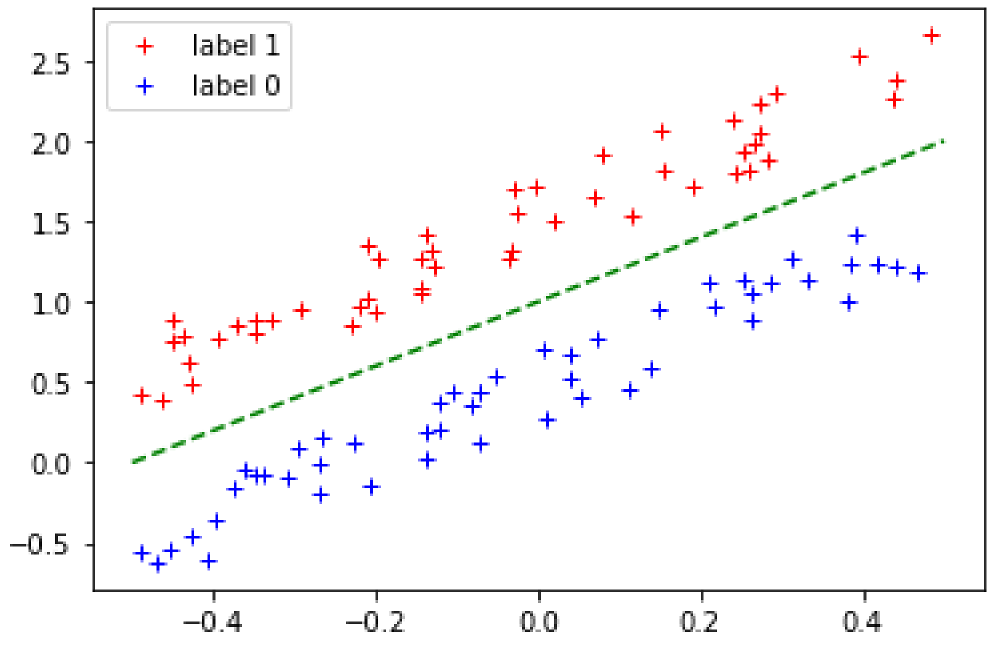
2/ Run the training with the default learning rate (alpha=1). Paste the plots with the situation at the start and with the situation at the end of the training in a text document. Paste also the start parameters (weight and bias) and trained parameters.

3/ Create a plot with the number of mis-classifications vs iteration.

1/ Prepare the dataset

[8]:
```python
m = 100
m1 = 50
a = 1
s = 2
x,y = prepare_data(m,m1,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)

print(x.shape)
print(y.shape)
```

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

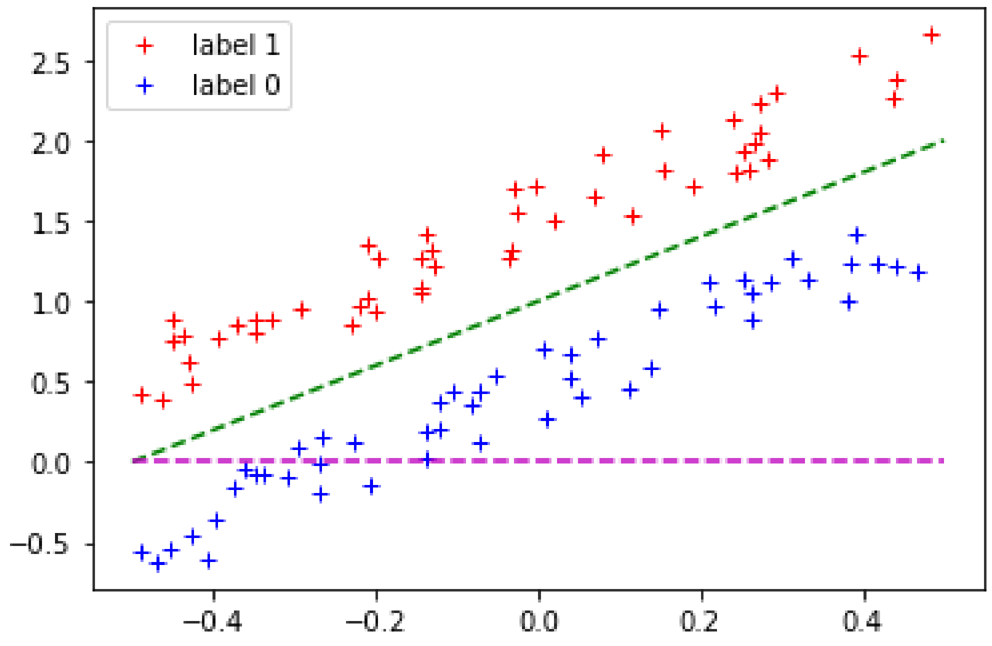


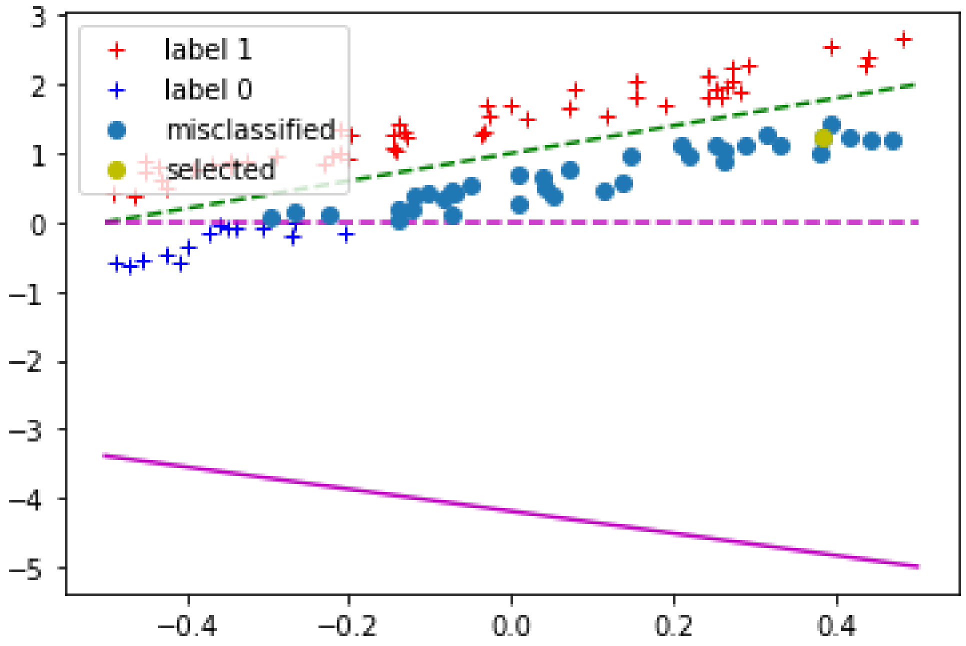
```
(2, 100)
(1, 100)
```

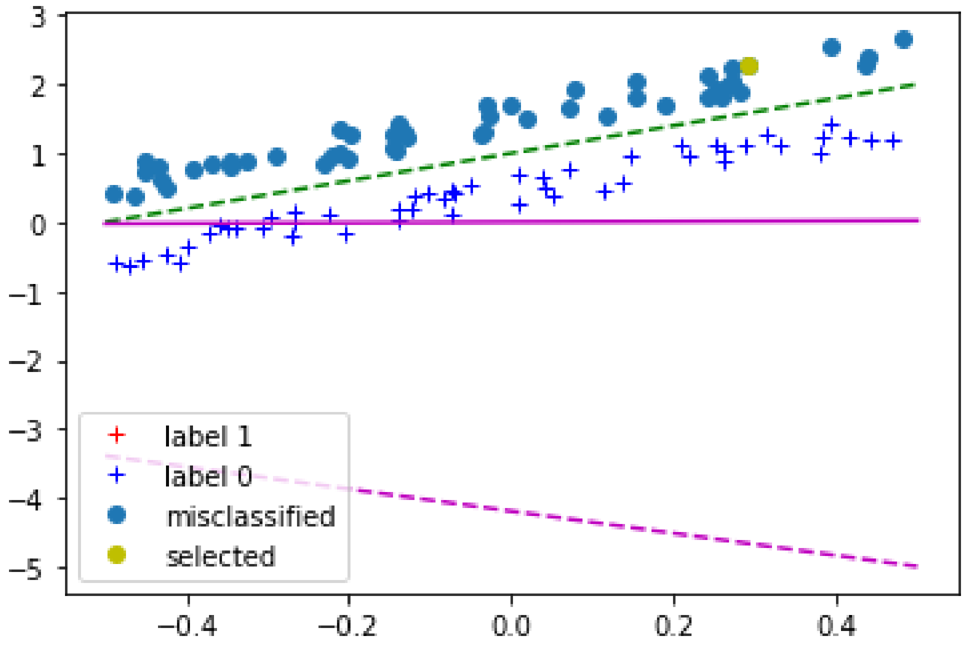
2/ Run the training

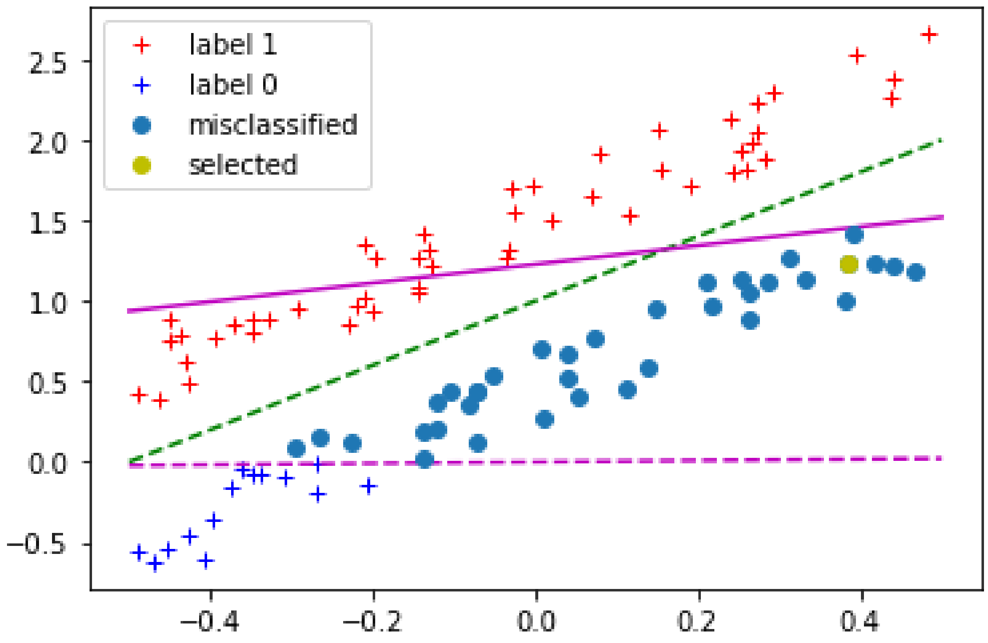
```
[9]: a1 = 0
s1 = 0
alpha = 1.0

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)
weight1,bias1,misclassified_counts = train(weight1, bias1, x, y, 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)
```

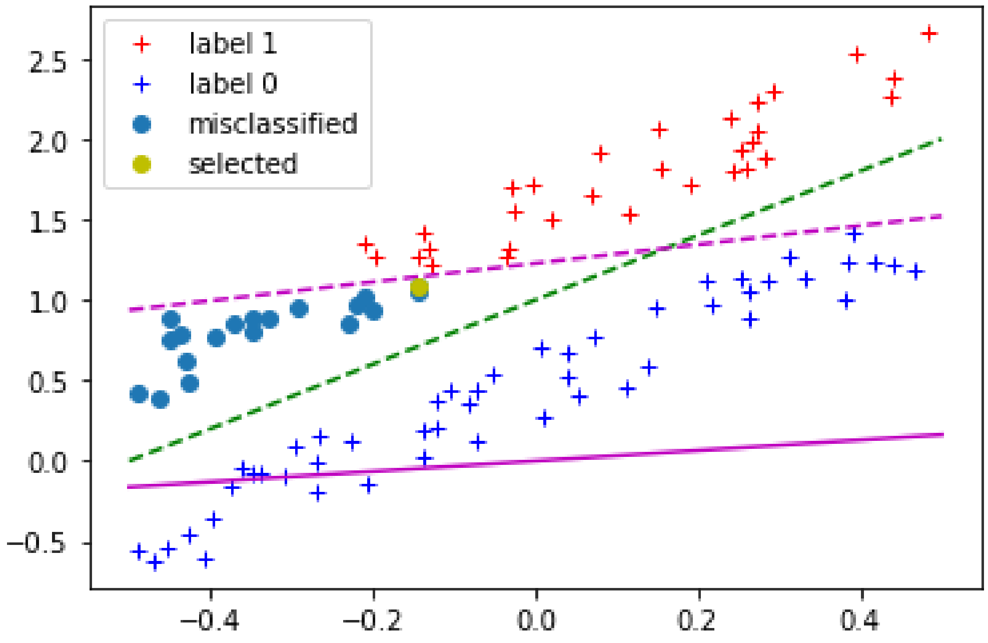
```
Initial Params:  [[0. 1.]] 0
```

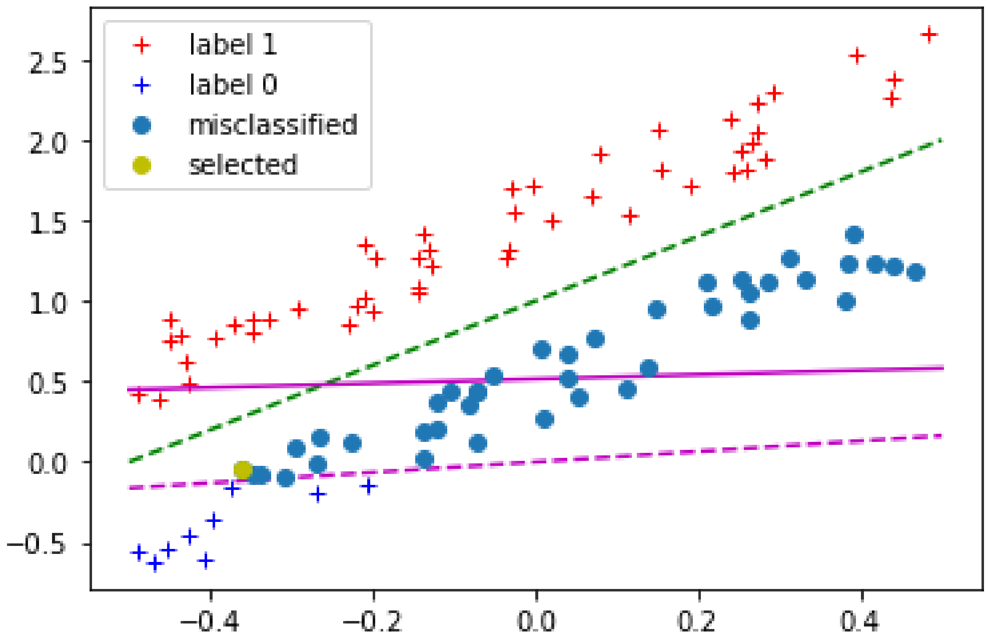


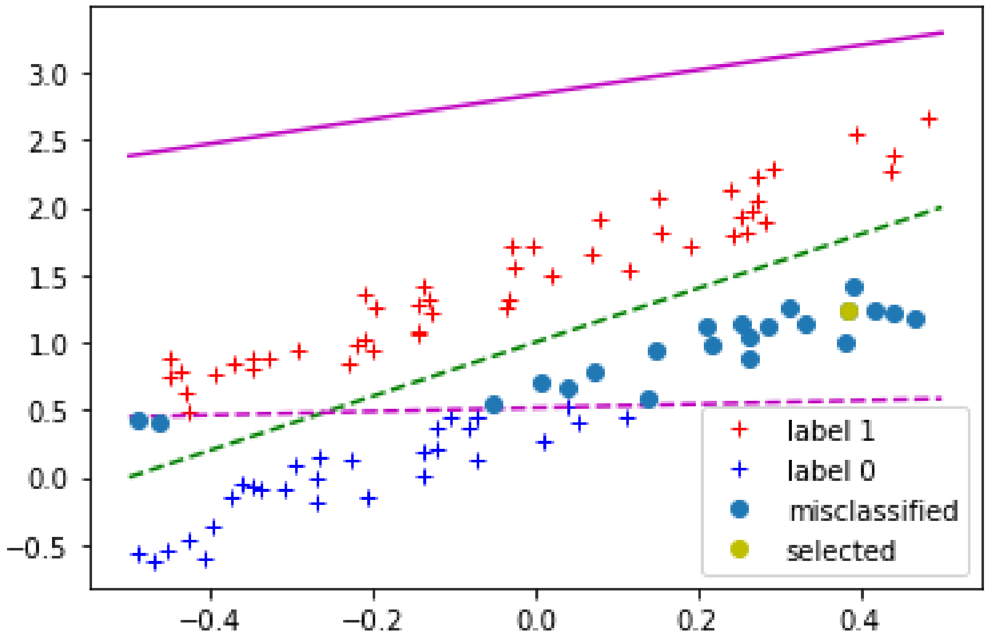

label 1
label 0
misclassified
selected
3
2
1
0
−1
−2
−3
−4
−5
−0.4
−0.2
0.0
0.2
0.4

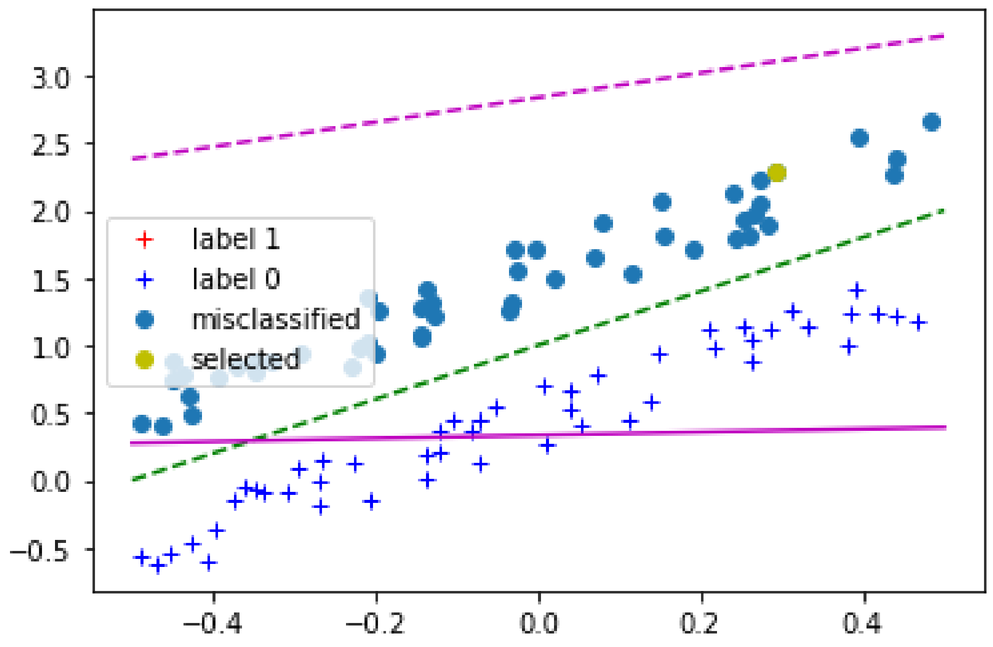

label 1
label 0
misclassified
selected
3
2
1
0
−1
−2
−3
−4
−5
−0.4
−0.2
0.0
0.2
0.4

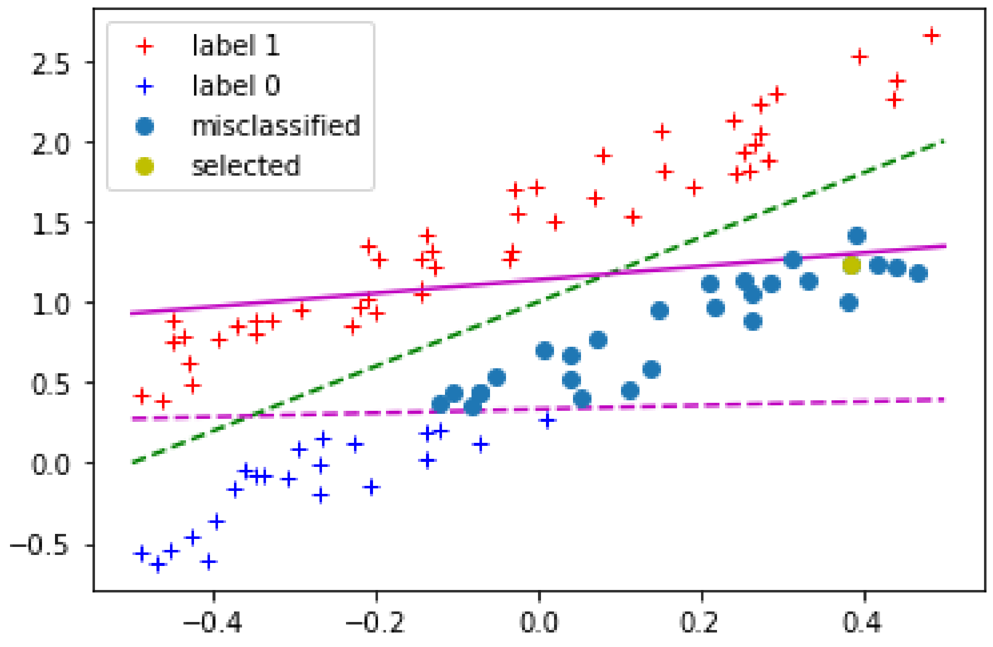

label 1
label 0
misclassified
selected
2.5
2.0
1.5
1.0
0.5
0.0
−0.5
−0.4
−0.2
0.0
0.2
0.4

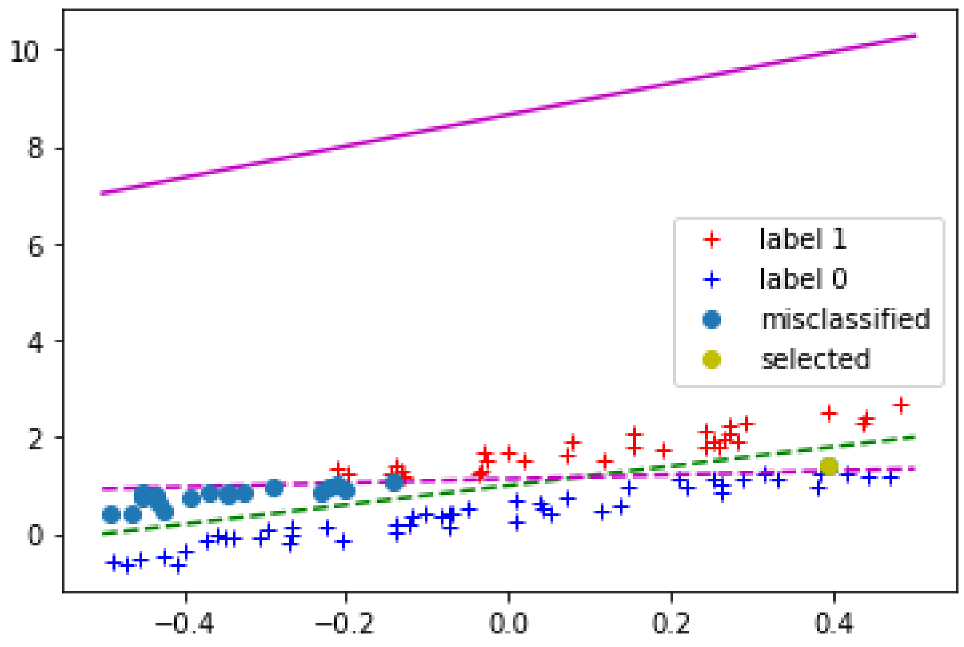

label 1
label 0
misclassified
selected
2.5
2.0
1.5
1.0
0.5
0.0
−0.5
−0.4
−0.2
0.0
0.2
0.4

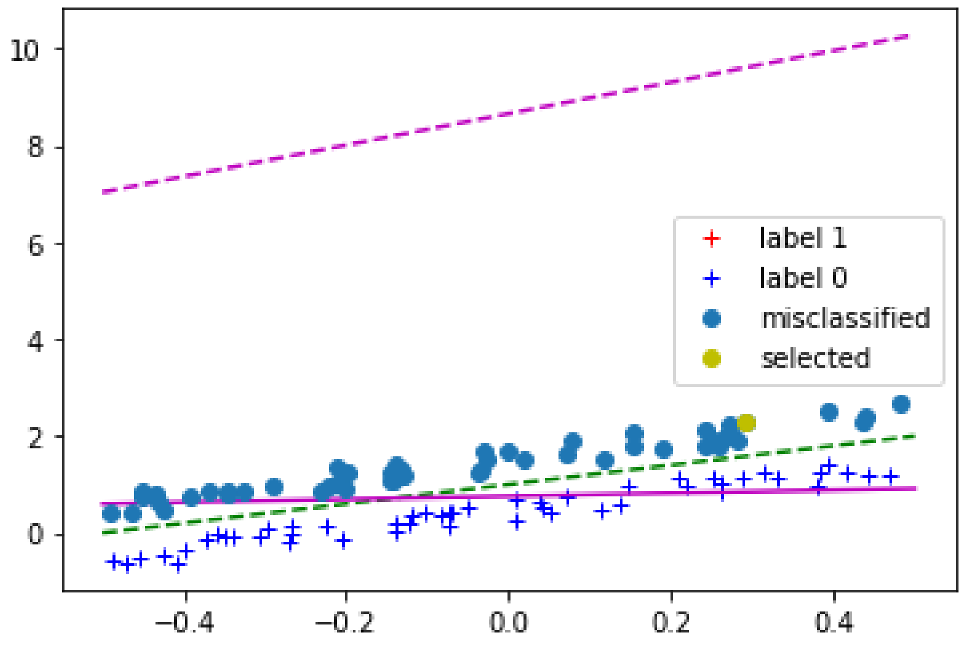

label 1
label 0
misclassified
selected
2.5
2.0
1.5
1.0
0.5
0.0
−0.5
−0.4
−0.2
0.0
0.2
0.4

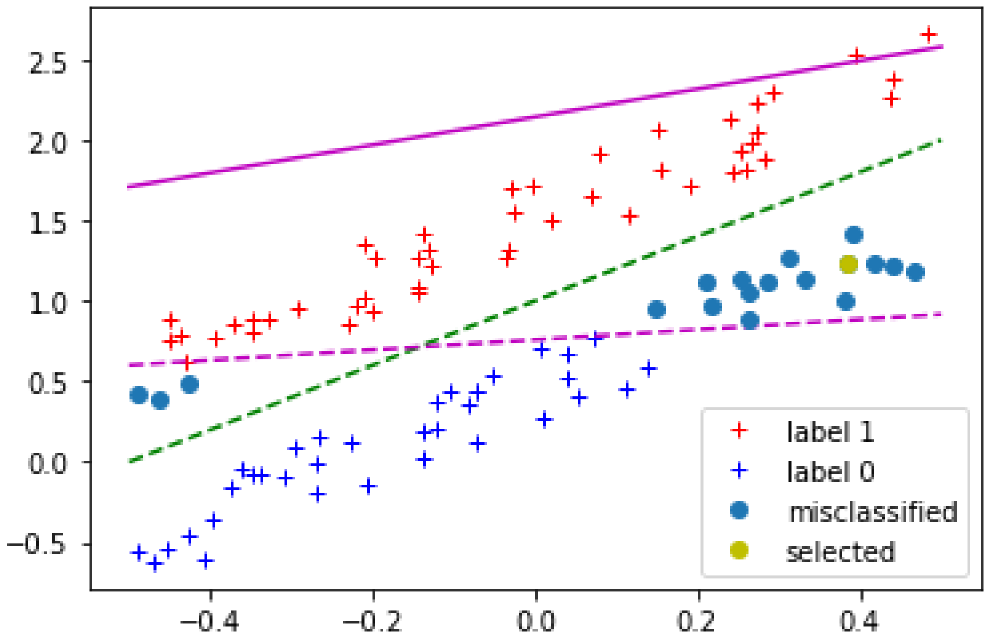

3.0
2.5
2.0
1.5
1.0
0.5
0.0
−0.5
label 1
label 0
misclassified
selected
−0.4
−0.2
0.0
0.2
0.4

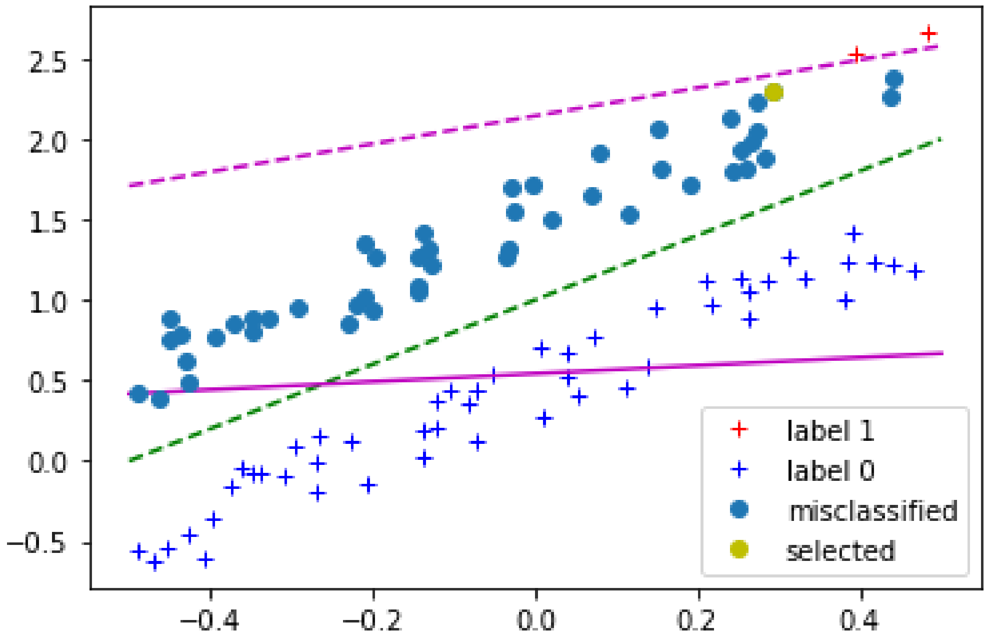

label 1
label 0
misclassified
selected

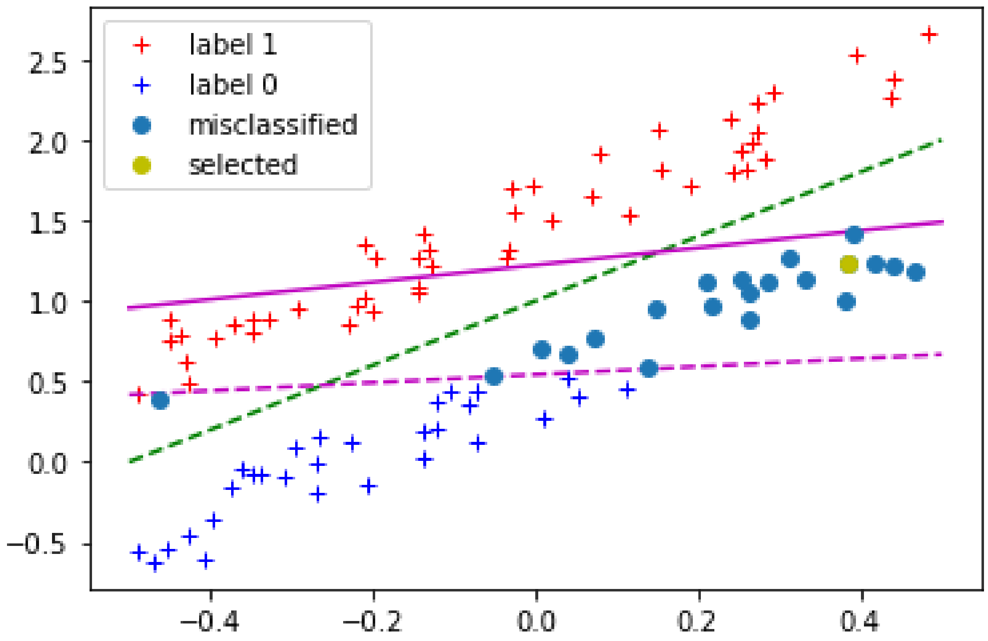

label 1
label 0
misclassified
selected

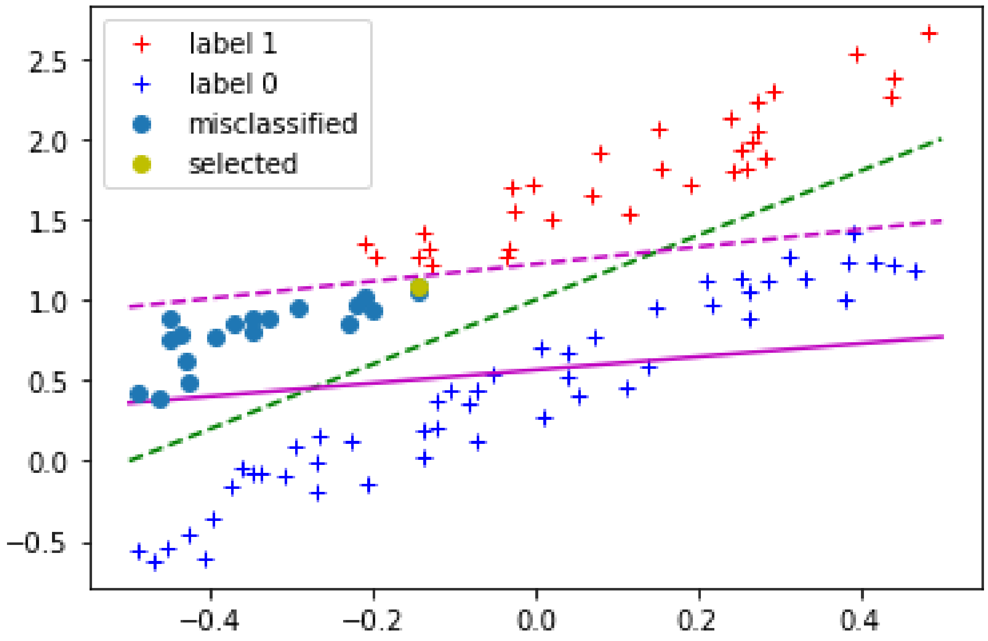

label 1
label 0
misclassified
selected
10
8
6
4
2
0
−0.4
−0.2
0.0
0.2
0.4

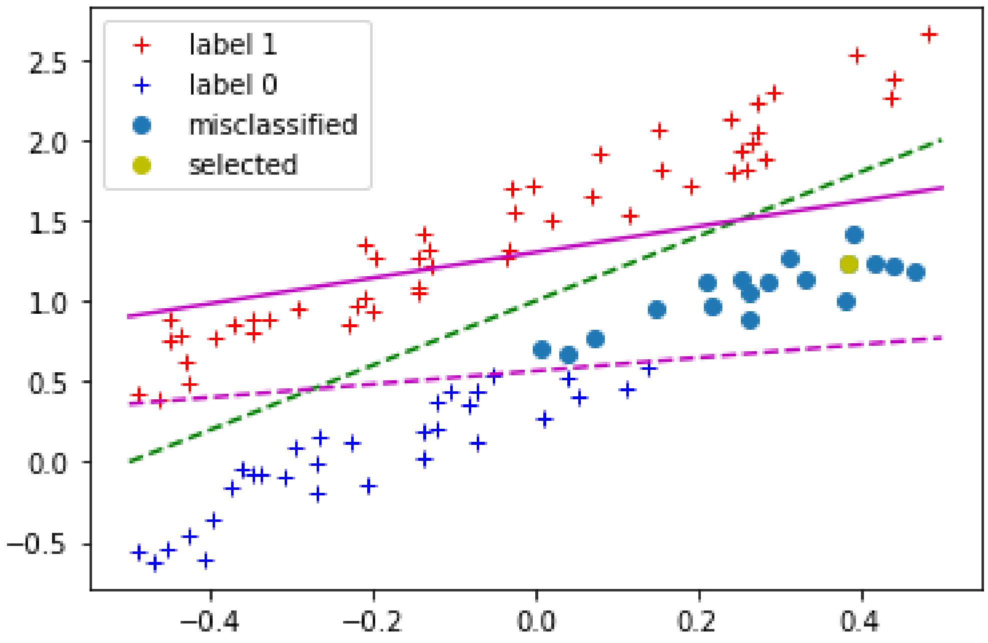

label 1
label 0
misclassified
selected
10
8
6
4
2
0
−0.4
−0.2
0.0
0.2
0.4

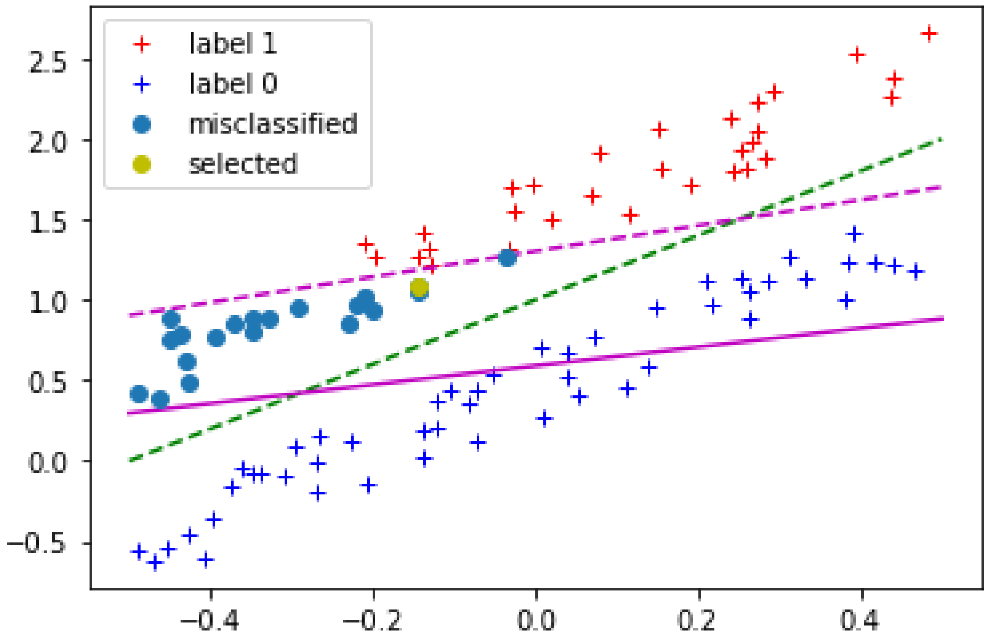

2.5
2.0
1.5
1.0
0.5
0.0
−0.5
−0.4
−0.2
0.0
0.2
0.4
label 1
label 0
misclassified
selected

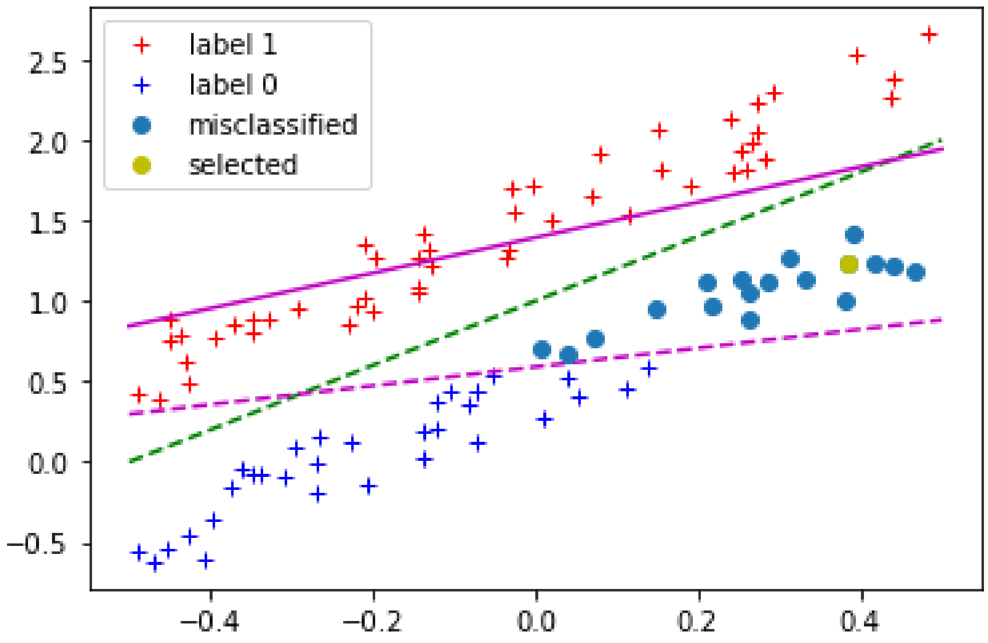

2.5
2.0
1.5
1.0
0.5
0.0
−0.5
−0.4
−0.2
0.0
0.2
0.4
label 1
label 0
misclassified
selected

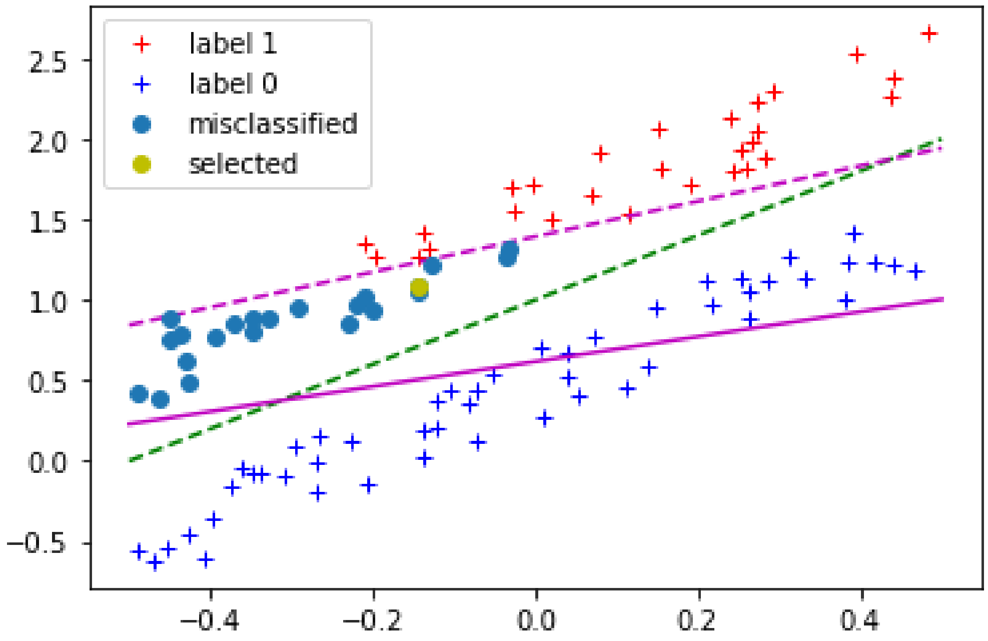

label 1
label 0
misclassified
selected
2.5
2.0
1.5
1.0
0.5
0.0
−0.5
−0.4
−0.2
0.0
0.2
0.4

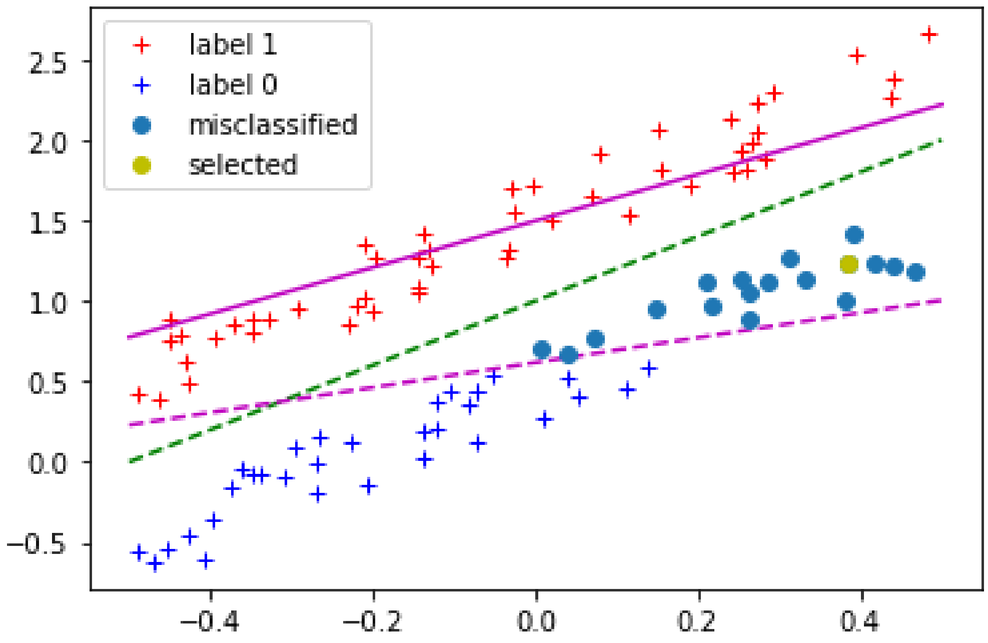

label 1
label 0
misclassified
selected
2.5
2.0
1.5
1.0
0.5
0.0
−0.5
−0.4
−0.2
0.0
0.2
0.4

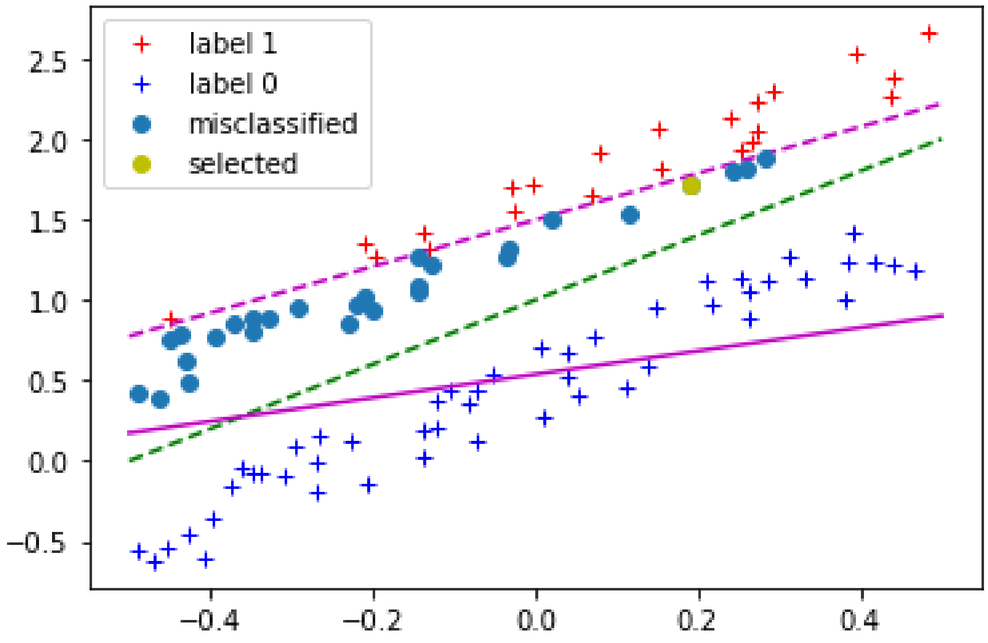

label 1
label 0
misclassified
selected
2.5
2.0
1.5
1.0
0.5
0.0
−0.5
−0.4
−0.2
0.0
0.2
0.4

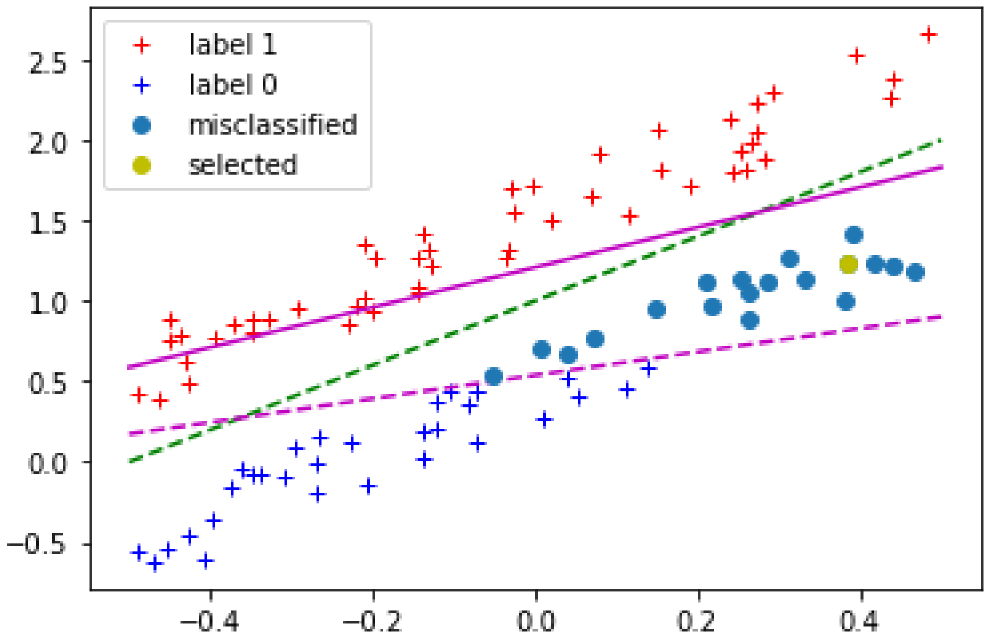

label 1
label 0
misclassified
selected
2.5
2.0
1.5
1.0
0.5
0.0
−0.5
−0.4
−0.2
0.0
0.2
0.4

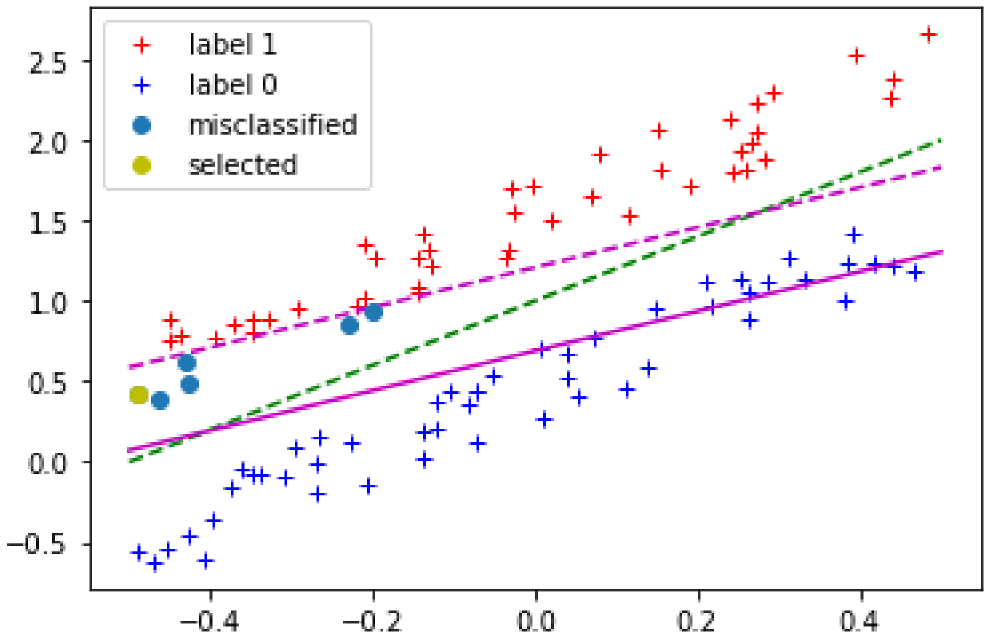

label 1
label 0
misclassified
selected
2.5
2.0
1.5
1.0
0.5
0.0
−0.5
−0.4
−0.2
0.0
0.2
0.4

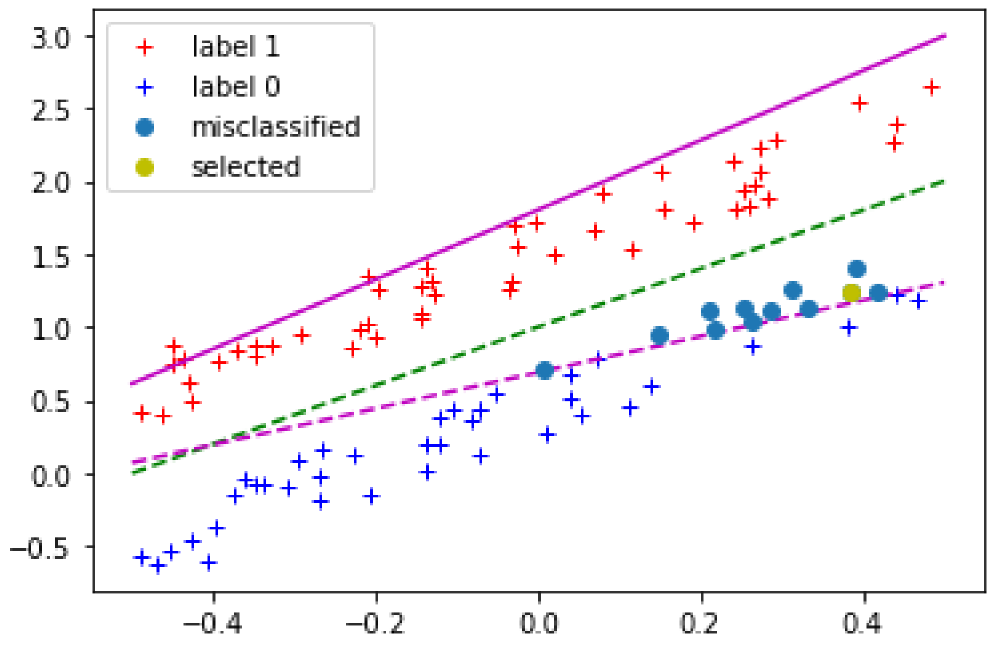

label 1
label 0
misclassified
selected
2.5
2.0
1.5
1.0
0.5
0.0
−0.5
−0.4
−0.2
0.0
0.2
0.4

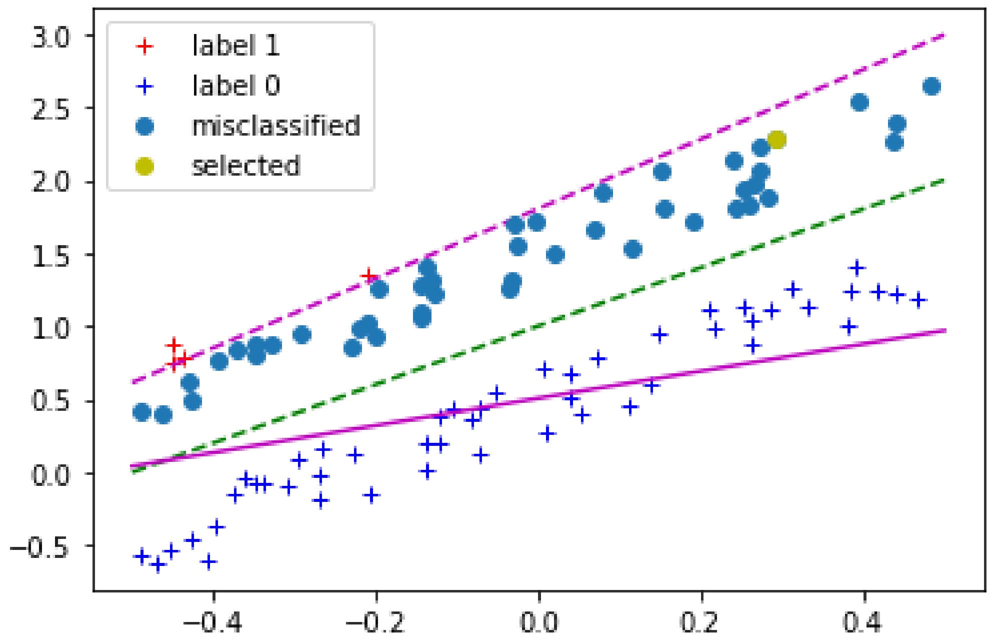

label 1
label 0
misclassified
selected
2.5
2.0
1.5
1.0
0.5
0.0
−0.5
−0.4
−0.2
0.0
0.2
0.4

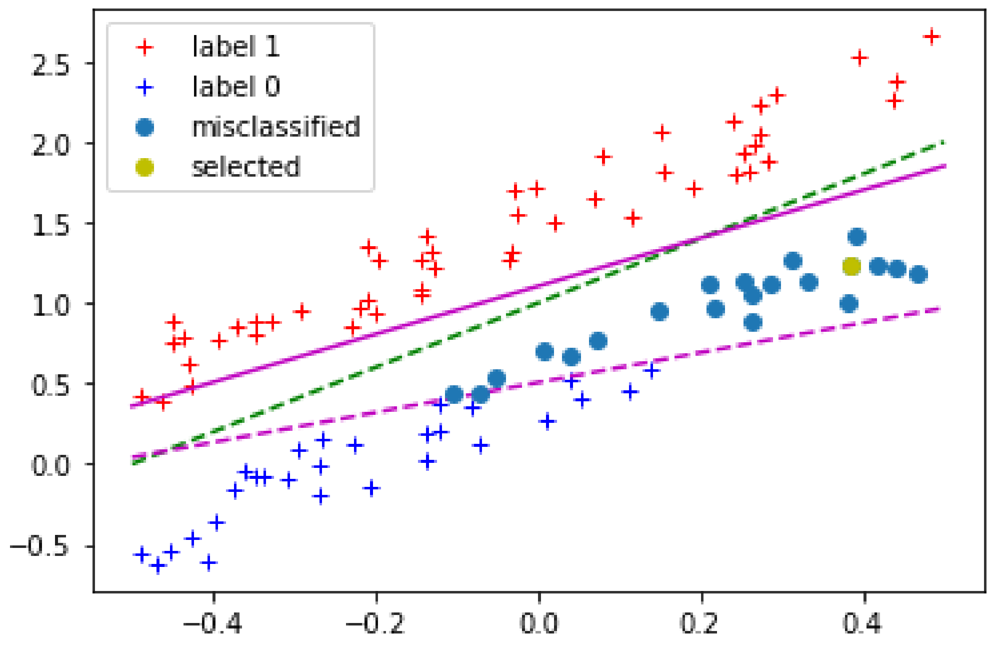

label 1
label 0
misclassified
selected
2.5
2.0
1.5
1.0
0.5
0.0
−0.5
−0.4
−0.2
0.0
0.2
0.4

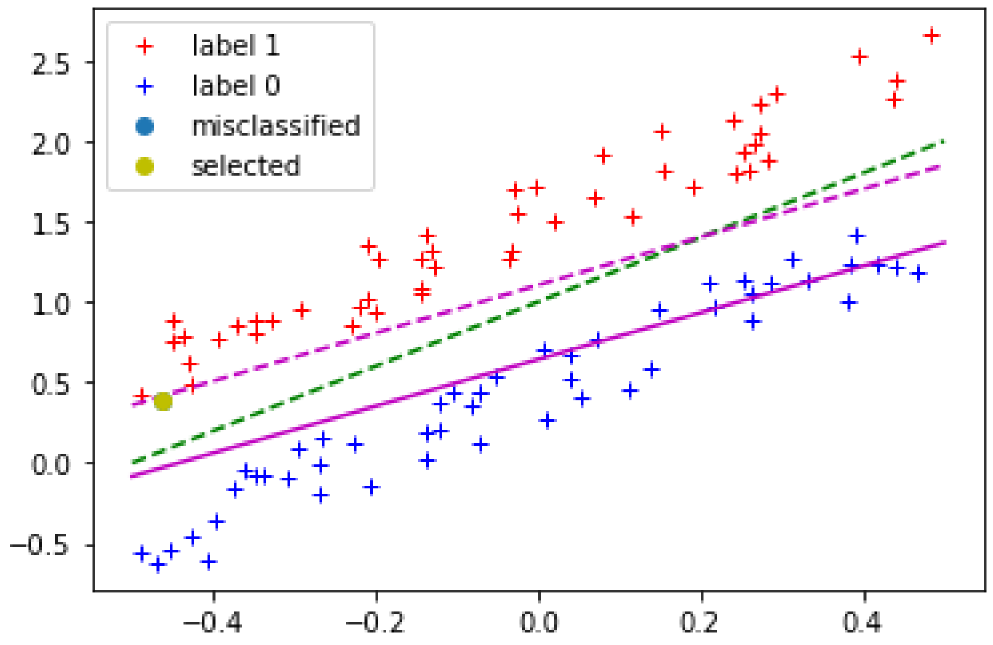

label 1
label 0
misclassified
selected
2.5
2.0
1.5
1.0
0.5
0.0
−0.5
−0.4
−0.2
0.0
0.2
0.4

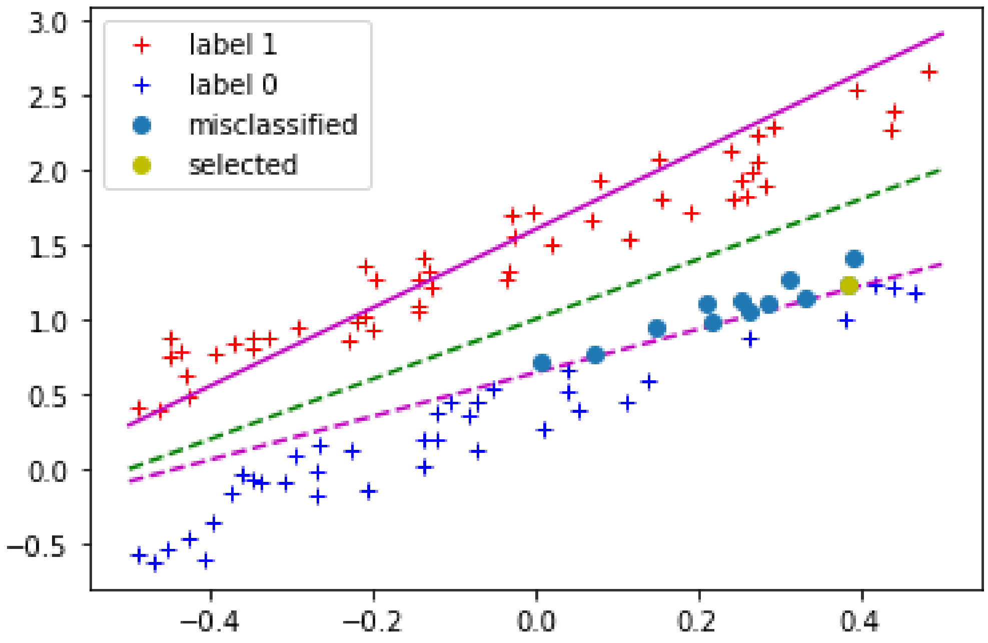

label 1
label 0
misclassified
selected
2.5
2.0
1.5
1.0
0.5
0.0
−0.5
−0.4
−0.2
0.0
0.2
0.4


label 1
label 0
misclassified
selected
3.0
2.5
2.0
1.5
1.0
0.5
0.0
−0.5
−0.4
−0.2
0.0
0.2
0.4

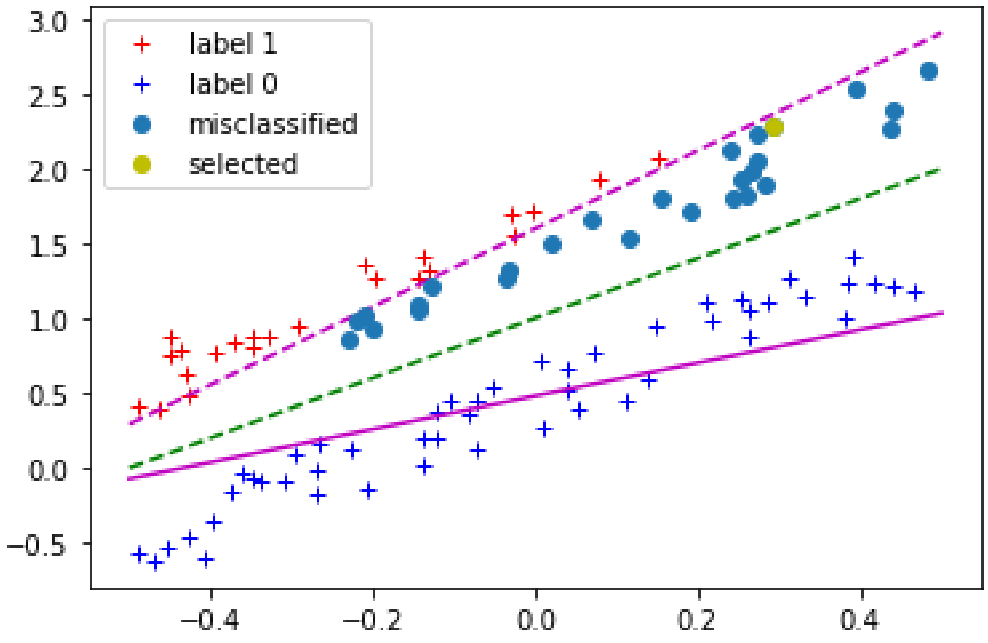

label 1
label 0
misclassified
selected
3.0
2.5
2.0
1.5
1.0
0.5
0.0
−0.5
−0.4
−0.2
0.0
0.2
0.4


label 1
label 0
misclassified
selected
2.5
2.0
1.5
1.0
0.5
0.0
−0.5
−0.4
−0.2
0.0
0.2
0.4

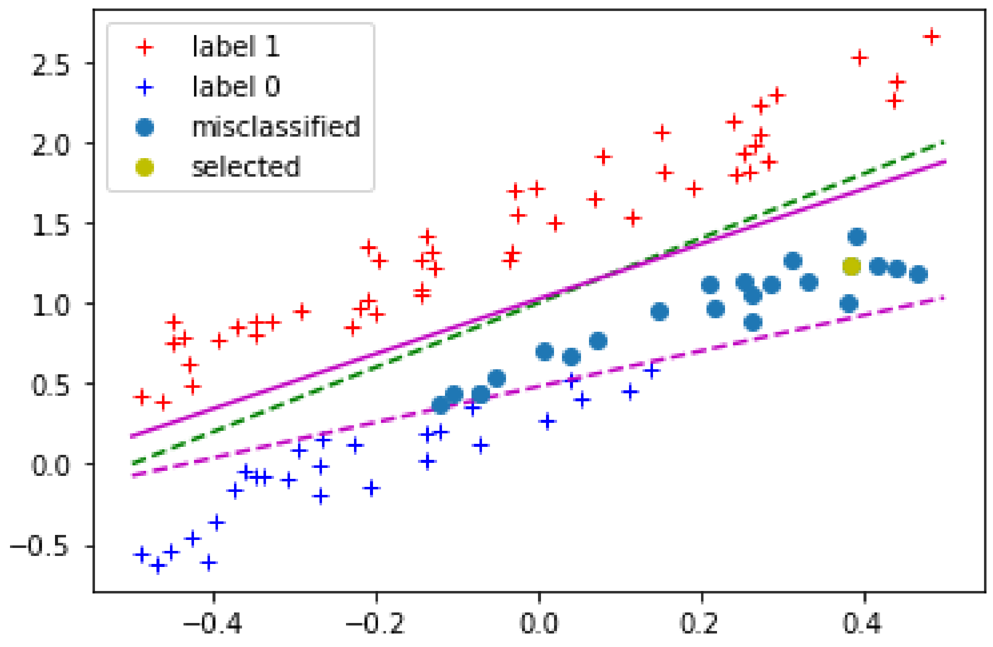

label 1
label 0
misclassified
selected
2.5
2.0
1.5
1.0
0.5
0.0
−0.5
−0.4
−0.2
0.0
0.2
0.4



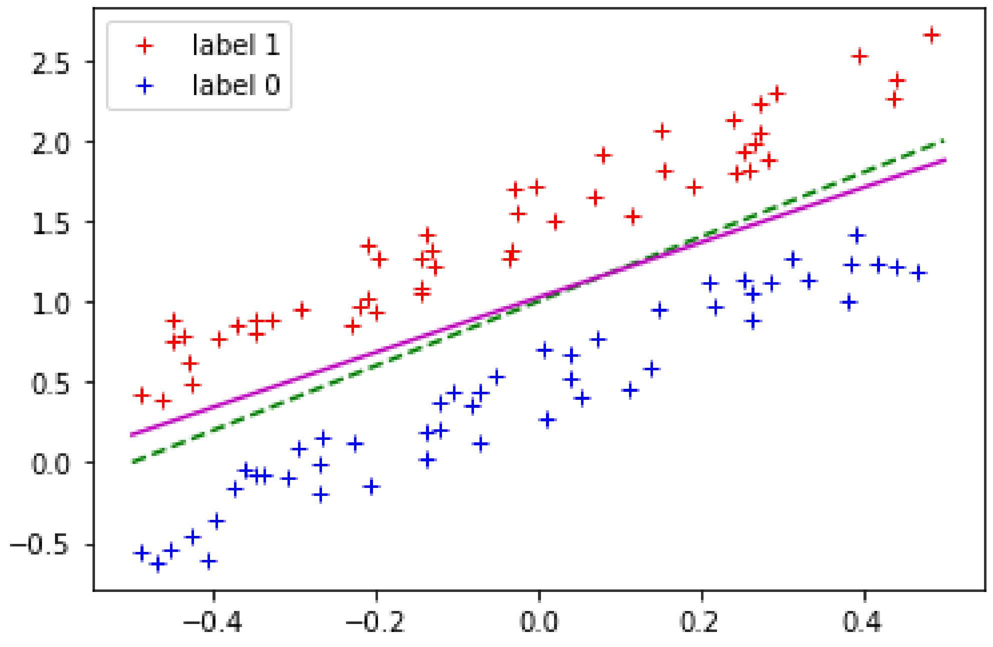
label 1
label 0
misclassified
selected



label 1
label 0
misclassified
selected



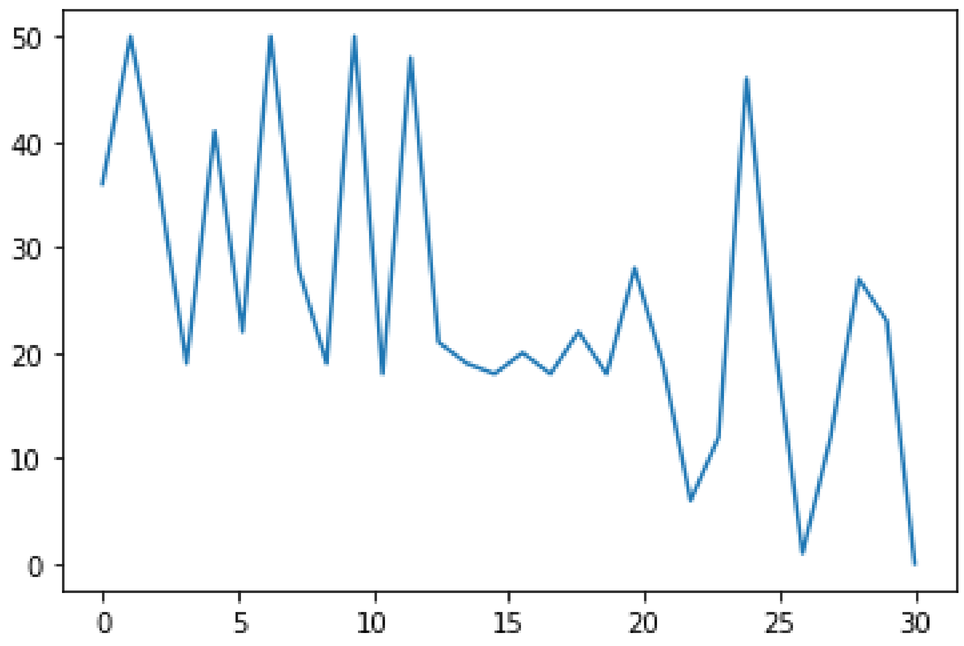
```
Iterations:  29
Trained Params:  [[-4.99046876  2.92867787]] [-3.]
```



3/ Create the plot with the misclassifications per iteration

[10]:
```python
nit = len(misclassified_counts)
it = np.linspace(0,nit,nit)

plt.plot(it, misclassified_counts)
```

[10]:
```
[<matplotlib.lines.Line2D at 0x7f16b63303a0>]
```

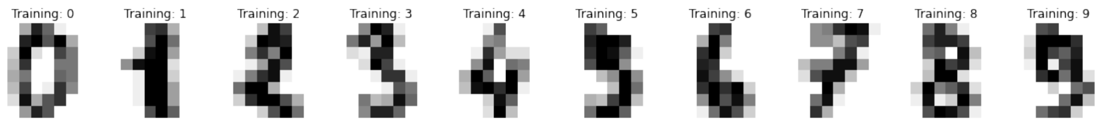


[ ]:
```python

```

0.2 Digits Dataset

https://scikit-learn.org/stable/auto_examples/classification/plot_digits_classification.html

[11]:
```python
from sklearn import datasets

digits = datasets.load_digits()

_, axes = plt.subplots(nrows=1, ncols=10, figsize=(20, 10))
for ax, image, label in zip(axes, digits.images, digits.target):
    ax.set_axis_off()
    ax.imshow(image, cmap=plt.cm.gray_r, interpolation='nearest')
    ax.set_title('Training: %i' % label)
```



[12]:
```python
n_samples = len(digits.images)
x_mnist = digits.images.reshape((n_samples, -1))
y_mnist = digits.target
print("X shape: ", x_mnist.shape)
print("Y shape: ", y_mnist.shape)
x_mnist
```

```
X shape:  (1797, 64)
Y shape:  (1797,)
```

[12]:
```
array([[ 0.,  0.,  5., …,  0.,  0.,  0.],
       [ 0.,  0.,  0., …, 10.,  0.,  0.],
       [ 0.,  0.,  0., …, 16.,  9.,  0.],
       …,
       [ 0.,  0.,  1., …,  6.,  0.,  0.],
       [ 0.,  0.,  2., …, 12.,  0.,  0.],
       [ 0.,  0., 10., …, 12.,  1.,  0.]])
```

[13]:
```python
mnist_pair_mask = (y_mnist == 0) | (y_mnist == 1)
x_mnist_pair = x_mnist[mnist_pair_mask]
y_mnist_pair = y_mnist[mnist_pair_mask]
print("X shape: ", x_mnist_pair.shape)
print("Y shape: ", y_mnist_pair.shape)
```

```
X shape:  (360, 64)
Y shape:  (360,)
```

0.2.1 Prepare the dataset in correct format for our algorithm

[14]:
```python
x_mnist_pair = x_mnist_pair.transpose()
y_mnist_pair = y_mnist_pair[np.newaxis, :]
print("X shape: ", x_mnist_pair.shape)
print("Y shape: ", y_mnist_pair.shape)
```

```
X shape:  (64, 360)
Y shape:  (1, 360)
```

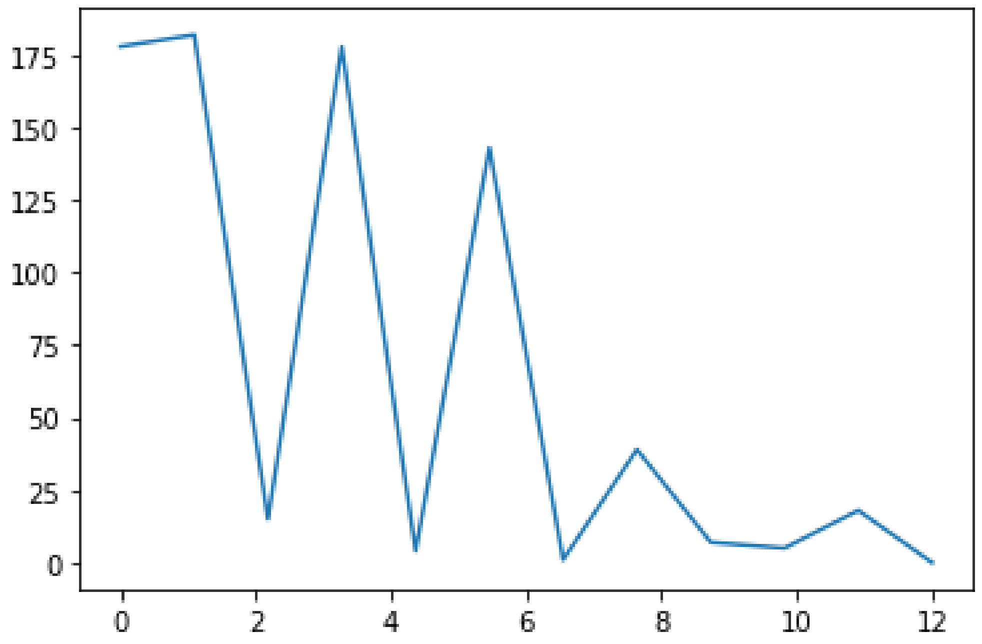
[15]:
```python
bias1_mnist = 1
weight1_mnist = np.full((1, 64), 1)

weight1_mnist,bias1_mnist,misclassified_counts = train(weight1_mnist,␣
 ↪bias1_mnist, x_mnist_pair, y_mnist_pair)
#weight1,bias1,misclassified_counts = train(weight1, bias1, x, y, debug=True,␣
 ↪params_best=params_best)
#params = lineparams(weight1, bias1)
```

```
print("Iterations: ", len(misclassified_counts)-1)
print("Trained Params: ", weight1_mnist,bias1_mnist)
```

```
Iterations:  11
Trained Params:  [[  1.    1.   -7. -15.    7.   22.    4.    1.    1.    1. -30. -10.
30.  -8.
   -6.    1.    1.    8. -26.   40.   64.   -7. -18.    1.    1.   13.   -2.   57.
   73.   -8. -23.    1.    1. -14. -38.   41.   59.   -3. -26.    1.    1.   -6.
  -48.    8.   41. -11. -16.    1.    1.   -1. -36. -14.    2. -13.    7.    1.
    1.    1.   -8. -19.   17.   40.    9.    1.]] [2.]
```

[16]:
```
nit = len(misclassified_counts)
it = np.linspace(0,nit,nit)

plt.plot(it, misclassified_counts)
```

[16]:
```
[<matplotlib.lines.Line2D at 0x7f16b3ebebb0>]
```



0.3 Search for pairs

[17]:
```
import itertools
from sklearn.utils import shuffle
```

[18]:
```
pair_results = []
for digit_pair in itertools.combinations(range(0,10), 2):
    d1, d2 = digit_pair
    mnist_pair_mask = (y_mnist == d1) | (y_mnist == d2)
    x_mnist_pair = x_mnist[mnist_pair_mask]
    y_mnist_pair = y_mnist[mnist_pair_mask]

    sample_num = x_mnist_pair.shape[0]

    x_mnist_pair, y_mnist_pair = shuffle(x_mnist_pair, y_mnist_pair,␣
↪random_state=0)

    x_mnist_pair = x_mnist_pair.transpose()
    y_mnist_pair = y_mnist_pair[np.newaxis, :]

    bias1_mnist = 1
    weight1_mnist = np.random.rand(1, 64)

    weight1_mnist,bias1_mnist,misclassified_counts = train(weight1_mnist,␣
↪bias1_mnist, x_mnist_pair, y_mnist_pair, alpha=0.001, max_iter=10000)
    #weight1,bias1,misclassified_counts = train(weight1, bias1, x, y,␣
↪debug=True, params_best=params_best)
    #params = lineparams(weight1, bias1)
    print("Pair:", digit_pair, "Iterations:", len(misclassified_counts)-1)
    # print("Trained Params: ", weight1_mnist,bias1_mnist)
    pair_results.append((digit_pair, sample_num, misclassified_counts))
```

```
Pair: (0, 1) Iterations: 44
Pair: (0, 2) Iterations: 10000
Pair: (0, 3) Iterations: 10000
Pair: (0, 4) Iterations: 10000
Pair: (0, 5) Iterations: 10000
Pair: (0, 6) Iterations: 10000
Pair: (0, 7) Iterations: 10000
Pair: (0, 8) Iterations: 10000
Pair: (0, 9) Iterations: 10000
Pair: (1, 2) Iterations: 10000
Pair: (1, 3) Iterations: 10000
Pair: (1, 4) Iterations: 10000
Pair: (1, 5) Iterations: 10000
Pair: (1, 6) Iterations: 10000
Pair: (1, 7) Iterations: 10000
Pair: (1, 8) Iterations: 10000
Pair: (1, 9) Iterations: 10000
Pair: (2, 3) Iterations: 10000
Pair: (2, 4) Iterations: 10000
Pair: (2, 5) Iterations: 10000
Pair: (2, 6) Iterations: 10000
Pair: (2, 7) Iterations: 10000
```

```
Pair: (2, 8) Iterations: 10000
Pair: (2, 9) Iterations: 10000
Pair: (3, 4) Iterations: 10000
Pair: (3, 5) Iterations: 10000
Pair: (3, 6) Iterations: 10000
Pair: (3, 7) Iterations: 10000
Pair: (3, 8) Iterations: 10000
Pair: (3, 9) Iterations: 10000
Pair: (4, 5) Iterations: 10000
Pair: (4, 6) Iterations: 10000
Pair: (4, 7) Iterations: 10000
Pair: (4, 8) Iterations: 10000
Pair: (4, 9) Iterations: 10000
Pair: (5, 6) Iterations: 10000
Pair: (5, 7) Iterations: 10000
Pair: (5, 8) Iterations: 10000
Pair: (5, 9) Iterations: 10000
Pair: (6, 7) Iterations: 10000
Pair: (6, 8) Iterations: 10000
Pair: (6, 9) Iterations: 10000
Pair: (7, 8) Iterations: 10000
Pair: (7, 9) Iterations: 10000
Pair: (8, 9) Iterations: 10000
```

[21]:
```
print("Pair\tAccuracy")
for digit_pair, sample_num, misclassified_counts in pair_results:
    accuracy = 1 - (misclassified_counts[-1] / sample_num)
    print("%s\t%0.2f" % (digit_pair, accuracy))
```

```
Pair    Accuracy
(0, 1)  1.00
(0, 2)  0.00
(0, 3)  0.10
(0, 4)  0.02
(0, 5)  0.11
(0, 6)  0.01
(0, 7)  0.27
(0, 8)  0.08
(0, 9)  0.04
(1, 2)  0.51
(1, 3)  0.50
(1, 4)  0.50
(1, 5)  0.50
(1, 6)  0.50
(1, 7)  0.50
(1, 8)  0.51
(1, 9)  0.50
(2, 3)  0.00
```

```
(2, 4)  0.00
(2, 5)  0.00
(2, 6)  0.00
(2, 7)  0.00
(2, 8)  0.00
(2, 9)  0.00
(3, 4)  0.00
(3, 5)  0.00
(3, 6)  0.00
(3, 7)  0.00
(3, 8)  0.00
(3, 9)  0.00
(4, 5)  0.00
(4, 6)  0.00
(4, 7)  0.00
(4, 8)  0.00
(4, 9)  0.00
(5, 6)  0.00
(5, 7)  0.00
(5, 8)  0.00
(5, 9)  0.00
(6, 7)  0.00
(6, 8)  0.00
(6, 9)  0.00
(7, 8)  0.00
(7, 9)  0.00
(8, 9)  0.00
```

```
[ ]:
```