Perceptron

Chapter Intended Learning Outcomes:

(i) Understand perceptron and its properties

(ii) Study linear model for classification and its relation with least squares, maximum likelihood estimation, and maximum *a posteriori* estimation

(iii) Study iterative methods and their properties as well as application in classification

(iv) Understand least-mean-square algorithm and its relation with perceptron

Classification

Classification refers to assigning a class or label to given input data:

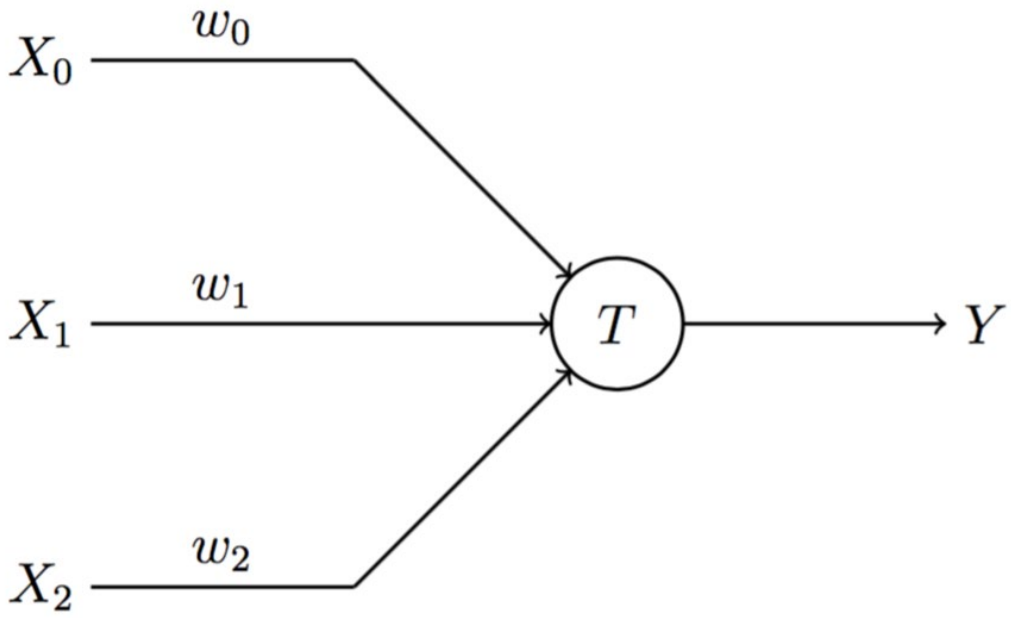
- Recognize a handwritten digit from the set of {0, 1, …, 9}.
- Recognize a person from an image database.
- Assign an input image as either 'cat' or 'dog'. This is called binary classification.
- Determine if an email is spam or non-spam. This can also be viewed as a binary hypothesis testing problem with null hypothesis (non-spam) and alternative hypothesis (spam).

This corresponds to supervised learning as pairs of inputs and outputs (labels) are required.

This is different from clustering where we divide the data into different unlabeled groups according to similarities in their features.

Brief Historical Development of Perceptron

McCulloch and Pitts proposed the first artificial neuron in 1940s as computing machine, e.g.:

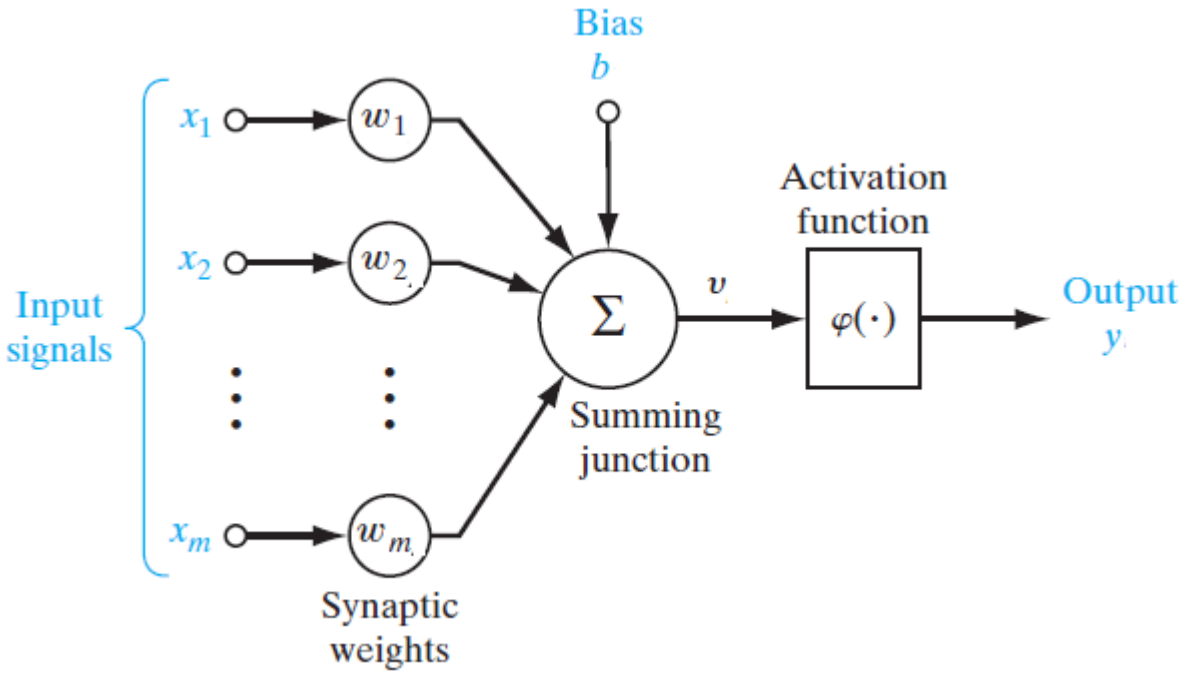


Input: $X_i \in \{0, 1\}$

Output: $$Y = \begin{cases} 1, & \sum_i w_i X_i \geq T \\ 0, & \text{otherwise} \end{cases}$$

Rosenblatt's perceptron generalized McCulloch-Pitts in 1950s which was the first supervised learning model.

Now the inputs are not restricted to be 0 or 1, but can be any real values, together with a bias term:



Perceptron which is a single artificial neuron, is able to perform the task of classifying 2 classes if they are linearly separable, i.e., they lie on the opposite sides of a hyperplane.

Properties and Algorithms for Perceptron

To perform classification, we need to determine $\{w_j\}$ and b using a set of training data of input-output pairs.

Given an input vector $\boldsymbol{x} = [x_1 \ \cdots \ x_m]^T$ to be classified, we first compute the linear combination:

$$v = \sum_{j=1}^{m} w_j x_j + b \tag{1}$$

then apply the activation function which is a sign function:

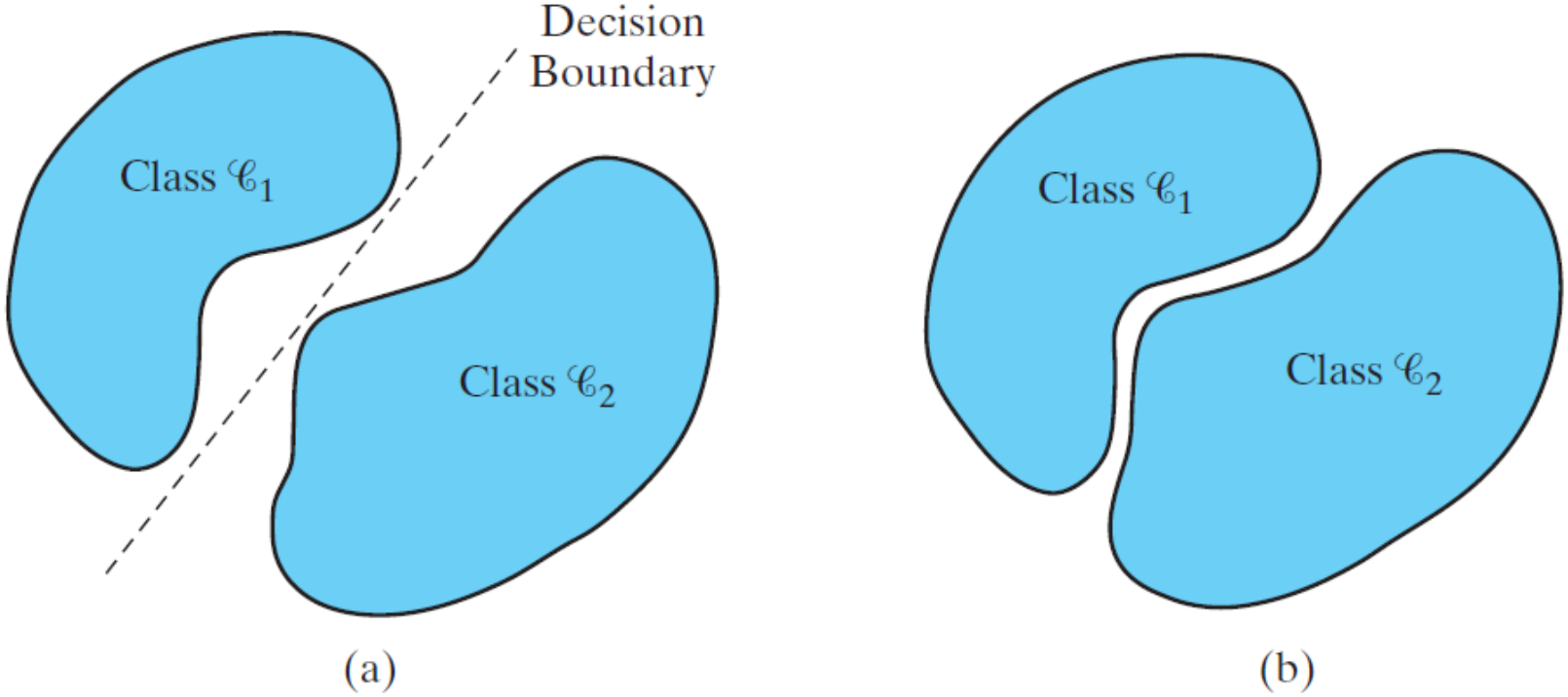
$$y = \text{sign}(v) = \begin{cases} 1, & v \geq 0 \\ -1, & v < 0 \end{cases} \tag{2}$$

That is, if v is positive, it corresponds to one class while it belongs to the other class if v is negative.

Hence the hyperplane, which is a subspace of one dimension less than its ambient space, is the decision boundary:

$$\sum_{j=1}^{m} w_j x_j + b = 0 \tag{3}$$

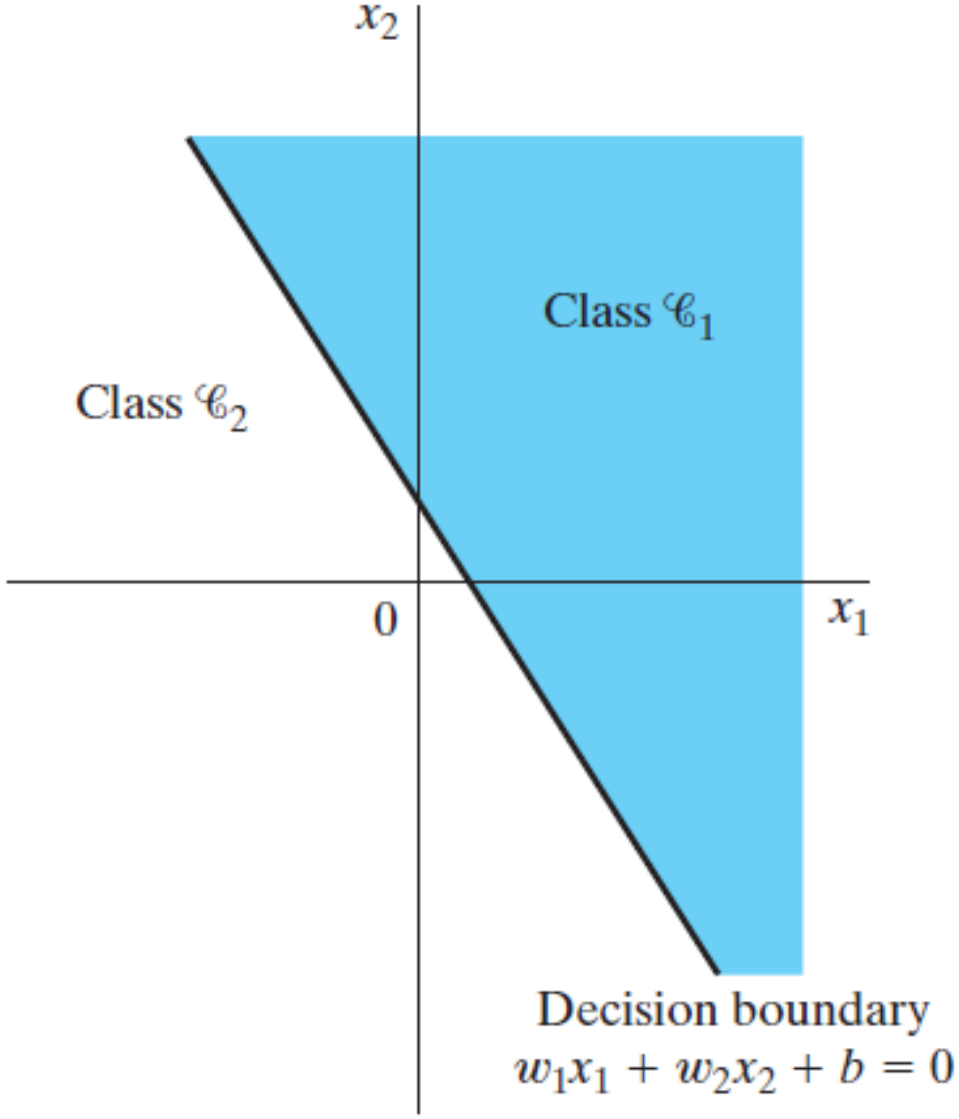
For 2-D input point $\boldsymbol{x} = [x_1 \; x_2]^T$, hyperplane is a line:



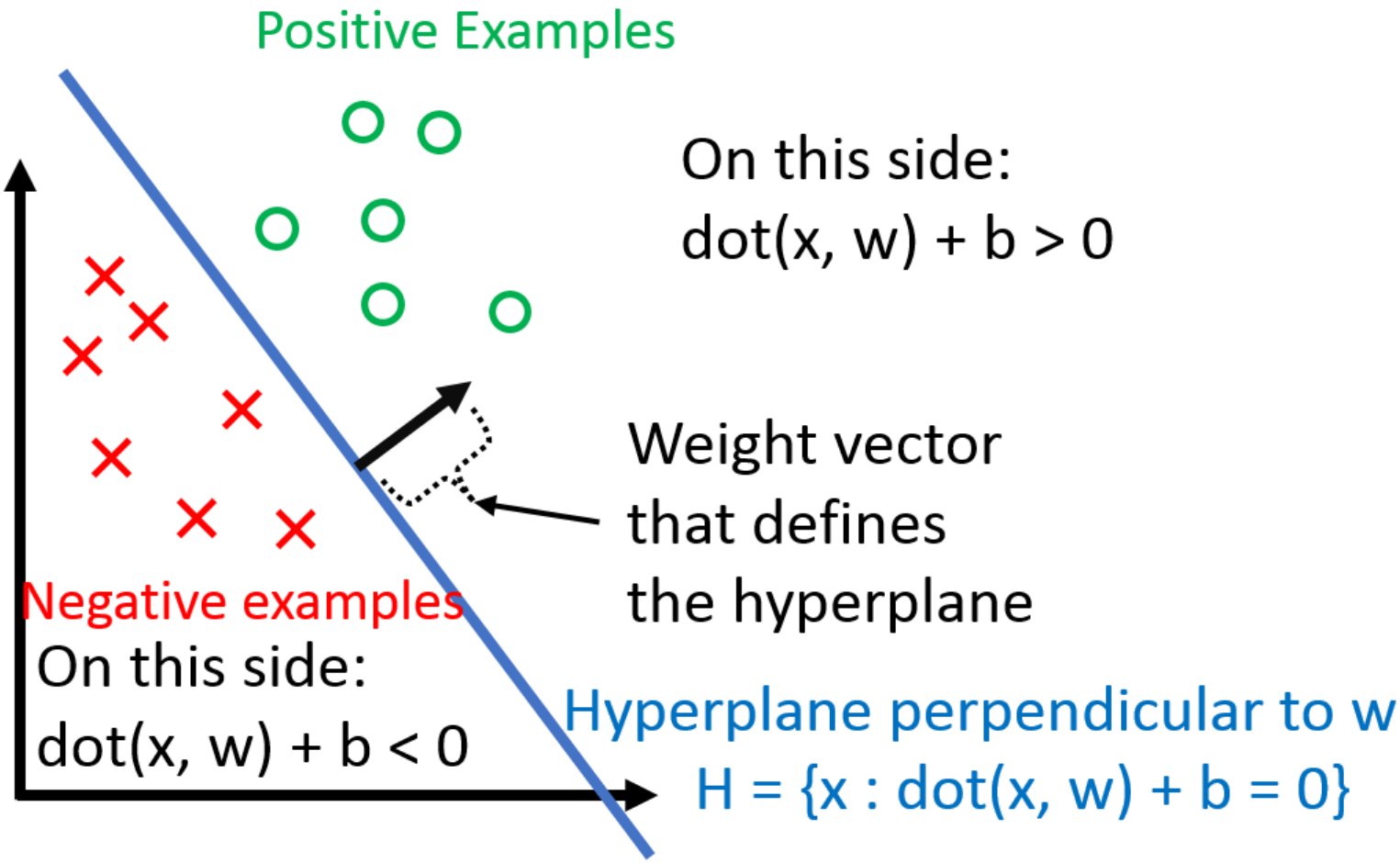
Linearly separable

Non-linearly separable

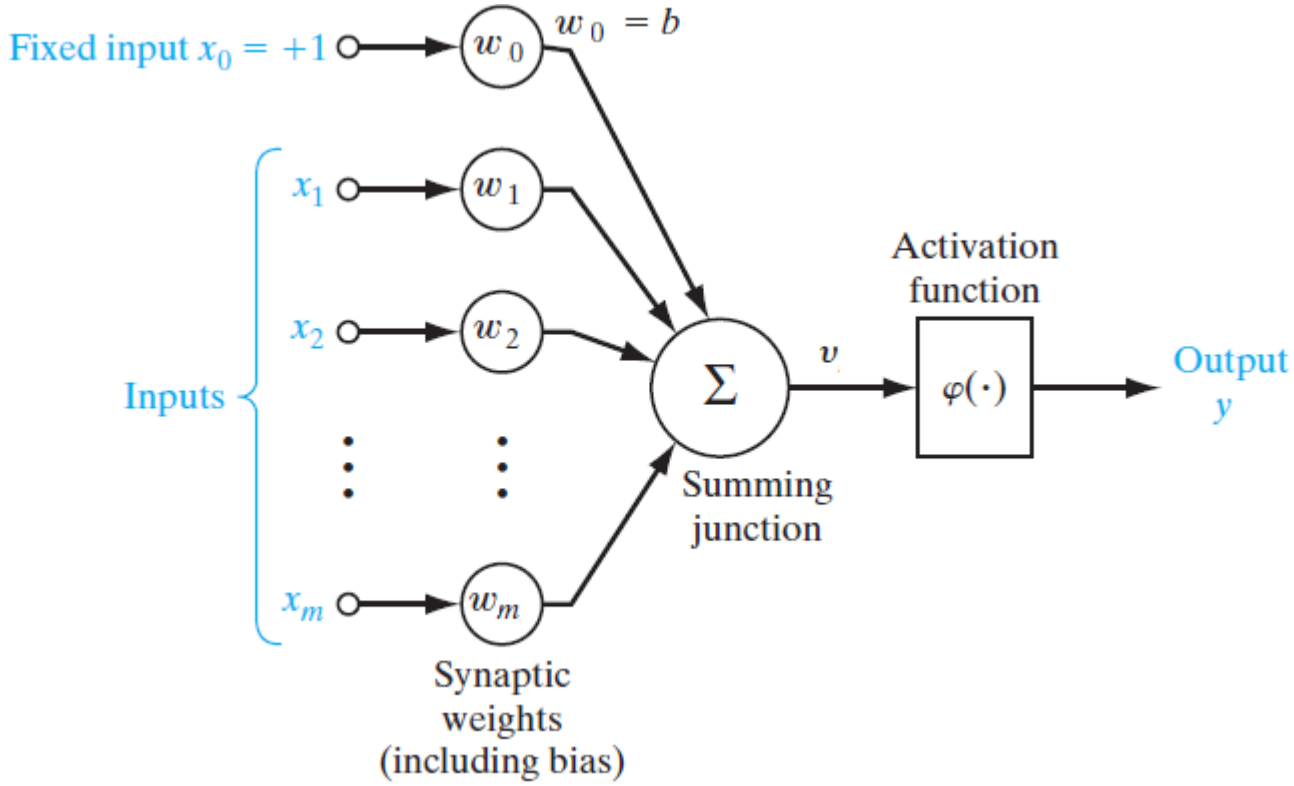
We can see that the bias provides a shift to the hyperplane, e.g., for 2-D case, b moves the straight line up or down



Geometrically, $[w_1\ w_2]^T$ is perpendicular to the hyperplane. If we consider $b = 0$, the hyperplane must pass through the origin as $[w_1\ w_2]\boldsymbol{x} = 0$.



It is more convenient to include the bias as a weight:



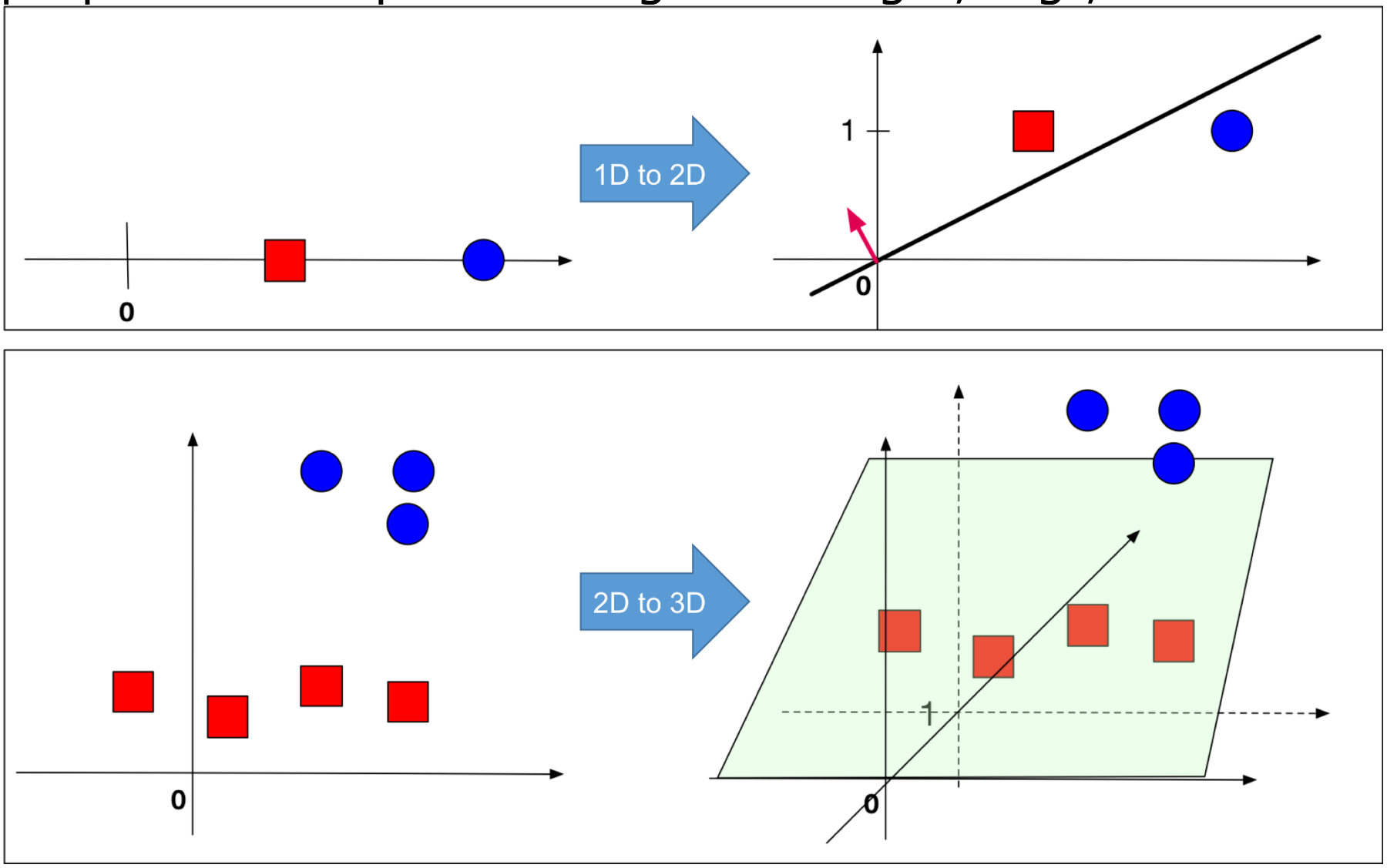
That is:

$$v = \boldsymbol{w}^T \boldsymbol{x} \tag{4}$$

where

$$\boldsymbol{x} = [x_0 \; x_1 \; \cdots \; x_m]^T, \quad x_0 = 1 \quad \text{and} \quad \boldsymbol{w} = [w_0 \; w_1 \; \cdots \; w_m]^T, \quad w_0 = b$$

With the addition of extra dimension of $x_0 = 1$, now the hyperplane must pass through the origin, e.g.,



Given a set of training vectors $\boldsymbol{x}_i$, $i = 1, \cdots, N$, with known labels, say, Classes $\mathcal{C}_1$ ($d_i = 1$) and $\mathcal{C}_2$ ($d_i = -1$), the basic idea to update $\boldsymbol{w}$ is:

If the ith input $\boldsymbol{x}_i$ is correctly classified by the weight vector at the tth iteration $\boldsymbol{w}_t$, no update for $\boldsymbol{w}_t$ in next iteration:

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t$$

if $\boldsymbol{w}_t^T \boldsymbol{x}_i \geq 0$ and $\boldsymbol{x}_i$ belongs to $\mathcal{C}_1$ or if $\boldsymbol{w}_t^T \boldsymbol{x}_i < 0$ and $\boldsymbol{x}_i$ belongs to $\mathcal{C}_2$.

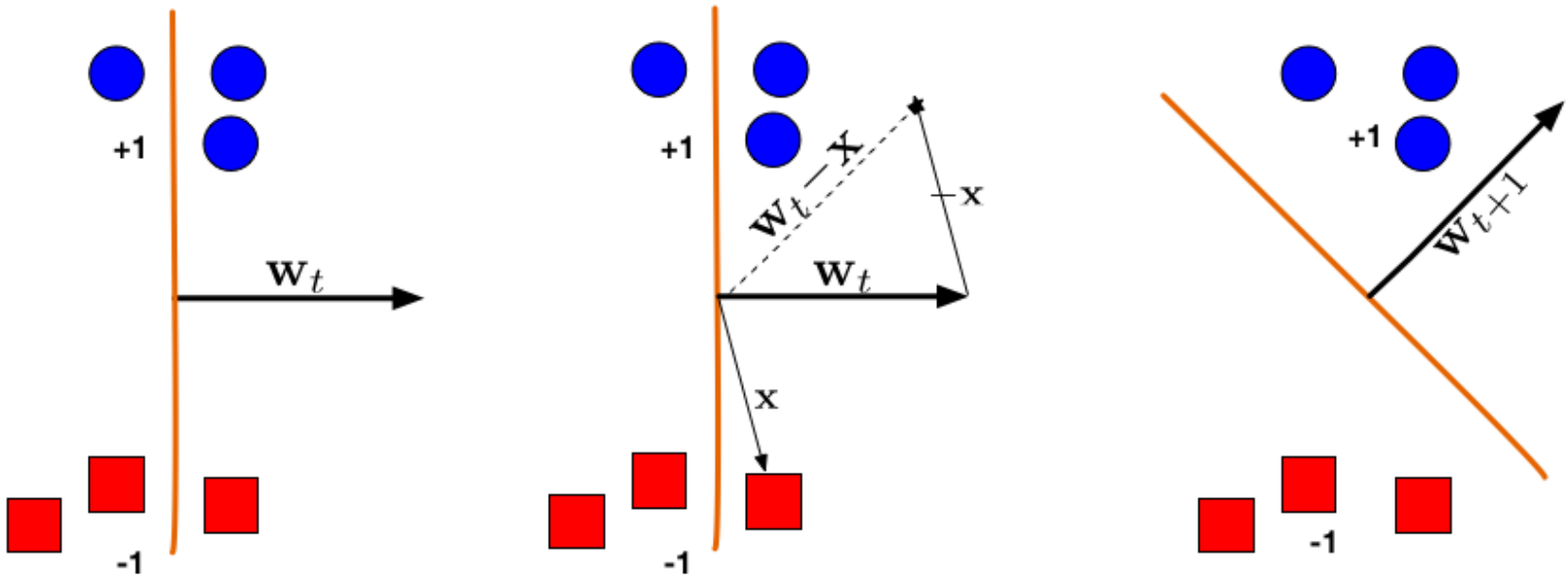
Otherwise, we update $\boldsymbol{w}_t$ in next iteration:

if $\boldsymbol{w}_t^T \boldsymbol{x}_i \geq 0$ and $\boldsymbol{x}_i$ belongs to $\mathcal{C}_2$: $\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \eta \boldsymbol{x}_i, \quad \eta > 0$

if $\boldsymbol{w}_t^T \boldsymbol{x}_i < 0$ and $\boldsymbol{x}_i$ belongs to $\mathcal{C}_1$: $\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \eta \boldsymbol{x}_i, \quad \eta > 0$

Here, η is the learning rate parameter which controls the adjustment at each iteration, and can be function of t.

Geometric intuition when an update is needed with $\eta = 1$:



Initializing $\boldsymbol{w}_0 = \mathbf{0}$, we use $\boldsymbol{x}_i$ in a cyclic manner, i.e.,

$$\{\boldsymbol{x}_1, \cdots, \boldsymbol{x}_N, \boldsymbol{x}_1, \cdots, \boldsymbol{x}_N, \boldsymbol{x}_1, \cdots\} = \{\boldsymbol{x}_1, \cdots, \boldsymbol{x}_t, \boldsymbol{x}_{t+1}, \cdots\}$$

to update $\boldsymbol{w}_t$ until all training samples are correctly classified. When a complete set of $\{\boldsymbol{x}_1, \cdots, \boldsymbol{x}_N\}$ goes through the algorithm, we refer it to as an epoch.

When $\boldsymbol{x}_t$ is correctly classified, this is equivalent to:

$$d_t = \text{sign}(\boldsymbol{w}_t^T \boldsymbol{x}_t) \Leftrightarrow d_t - \text{sign}(\boldsymbol{w}_t^T \boldsymbol{x}_t) = 0 \Leftrightarrow d_t(\boldsymbol{w}_t^T \boldsymbol{x}_t) > 0 \tag{5}$$

This implies when $\boldsymbol{x}_t$ is incorrectly classified:

$$d_t - \text{sign}(\boldsymbol{w}_t^T \boldsymbol{x}_t) = \begin{cases} -2, & d_t = -1,\ \text{sign}(\boldsymbol{w}_t^T \boldsymbol{x}_t) = 1 \\ 2, & d_t = 1,\ \text{sign}(\boldsymbol{w}_t^T \boldsymbol{x}_t) = -1 \end{cases} \tag{6}$$

The perceptron algorithm can be written in compact form as:

1. Initialize $\boldsymbol{w}_0 = \boldsymbol{0}$
2. For $t = 1, 2, \cdots$

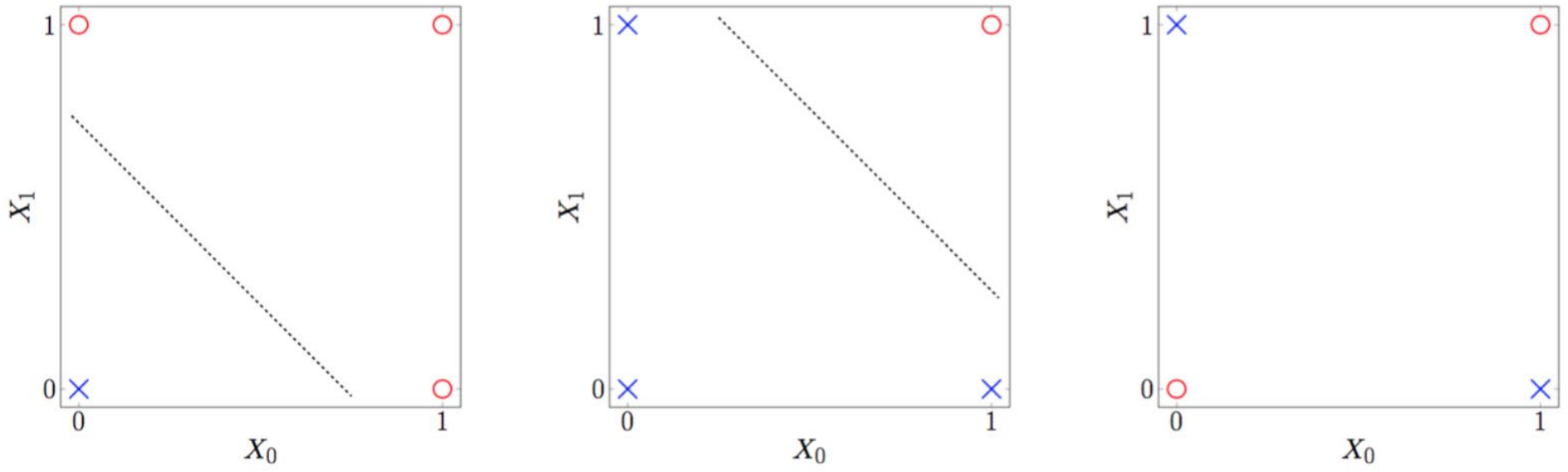
   Update the weight vector as:

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + 0.5\eta[d_t - \text{sign}(\boldsymbol{w}_t^T \boldsymbol{x}_t)]\boldsymbol{x}_t \tag{7}$$

   until there is no change in $\boldsymbol{w}_t$ for all training samples.

As long as the two classes are linearly separable, $\boldsymbol{w}_t$ must converge after finite number of iterations.

Note that the algorithm can also be implemented in different forms, e.g., updating $\boldsymbol{w}_t$ if $d_t(\boldsymbol{w}_t^T \boldsymbol{x}_t) < 0$.



(a) OR (b) AND (c) XOR

Can perceptron realize OR function? How about AND function? And how about XOR function? Why?
Is the hyperplane in perceptron unique?

Convergence Proof of Perceptron

When the dataset is linearly separable, the perceptron will find a separating hyperplane in a finite number of updates. Otherwise, $\boldsymbol{w}_t$ will not converge.

Suppose a hyperplane characterized by $\boldsymbol{w}^\star$ exists. This means that $d_i(\boldsymbol{x}_i^T \boldsymbol{w}^\star) > 0$ for the entire dataset.

Without loss of generality, we assume:

$$\eta = 1$$

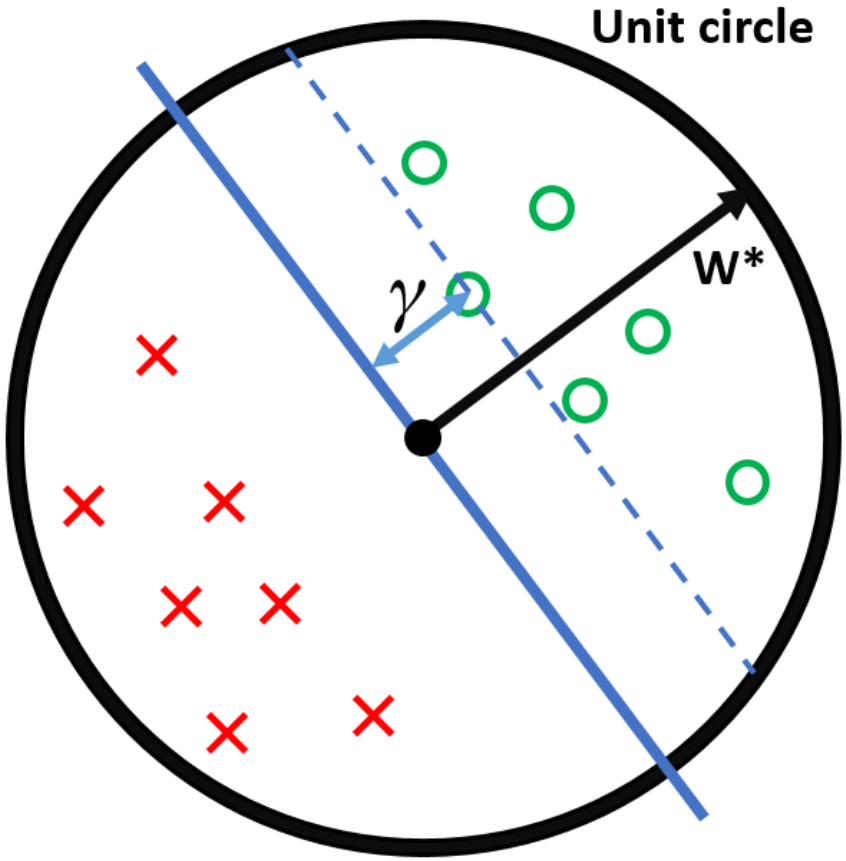
$$\|\boldsymbol{w}^\star\|_2 = 1$$

and

$$\|\boldsymbol{x}_i\|_2 \leq 1$$

We also define:

$$\gamma = \min |\boldsymbol{x}_i^T \boldsymbol{w}^\star|$$



Under the above settings, the perceptron algorithm makes at most $1/\gamma^2$ mistakes.

Proof:

To ease the presentation, all subscripts are removed.

Recall

$$d(\boldsymbol{x}^T\boldsymbol{w}^\star) > 0$$

When an update is needed, i.e., $\boldsymbol{w} \leftarrow \boldsymbol{w} + d\boldsymbol{x}$, we know $d(\boldsymbol{x}^T\boldsymbol{w}) < 0$.

Now we consider this effect on two terms: $\boldsymbol{w}^T\boldsymbol{w}^\star$ and $\boldsymbol{w}^T\boldsymbol{w}$.

For $\boldsymbol{w}^T\boldsymbol{w}^\star$, the effect is:

$$(\boldsymbol{w} + d\boldsymbol{x})^T\boldsymbol{w}^\star = \boldsymbol{w}^T\boldsymbol{w}^\star + d(\boldsymbol{x}^T\boldsymbol{w}^\star) \geq \boldsymbol{w}^T\boldsymbol{w}^\star + \gamma$$

because

$$d(\boldsymbol{x}^T\boldsymbol{w}^\star) = |\boldsymbol{x}^T\boldsymbol{w}^\star| \geq \gamma$$

This means that for each update, $\boldsymbol{w}^T\boldsymbol{w}^\star$ grows by at least γ.

For $\boldsymbol{w}^T\boldsymbol{w}$, the effect is:

$$(\boldsymbol{w}+d\boldsymbol{x})^T(\boldsymbol{w}+d\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{w} + \underbrace{2d(\boldsymbol{w}^T\boldsymbol{x})}_{<0} + \underbrace{d^2(\boldsymbol{x}^T\boldsymbol{x})}_{0\leq\ \leq 1} \leq \boldsymbol{w}^T\boldsymbol{w}+1$$

because

$$d(\boldsymbol{w}^T\boldsymbol{x}) < 0, \quad d^2 = 1, \qquad \|\boldsymbol{x}\|_2^2 \leq 1$$

This means that for each update, $\boldsymbol{w}^T\boldsymbol{w}$ grows by at most 1.

Hence after certain number of updates, say, M, the following two inequalities must hold:

$$\boldsymbol{w}^T\boldsymbol{w}^\star \geq M\gamma \tag{8}$$

$$\boldsymbol{w}^T\boldsymbol{w} \leq M \tag{9}$$

With the use of (8)-(9), we complete the proof as follows:

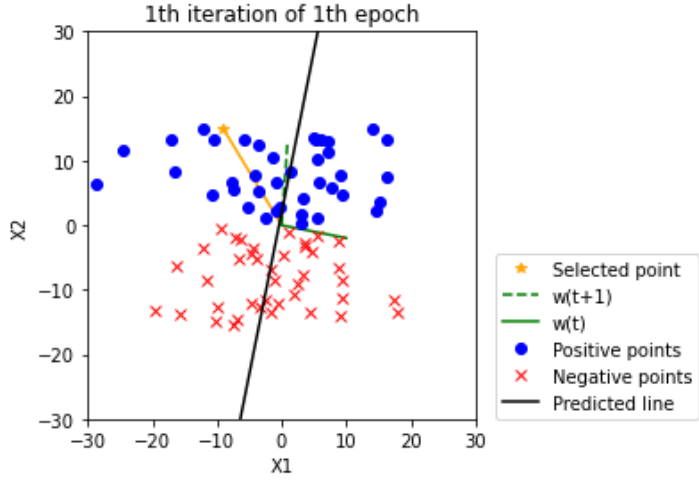
$$
\begin{aligned}
M\gamma &\leq \boldsymbol{w}^T\boldsymbol{w}^\star \\
&= \|\boldsymbol{w}\|_2\|\boldsymbol{w}^\star\|_2\cos(\theta), \quad \theta \text{ is angle between } \boldsymbol{w} \text{ and } \boldsymbol{w}^\star \\
&= \|\boldsymbol{w}\|_2\cos(\theta), \quad \|\boldsymbol{w}^\star\|_2 = 1 \\
&\leq \|\mathbf{w}\|_2, \quad \cos(\theta) \leq 1 \\
&= \sqrt{\boldsymbol{w}^T\boldsymbol{w}} \\
&\leq \sqrt{M}
\end{aligned}
$$

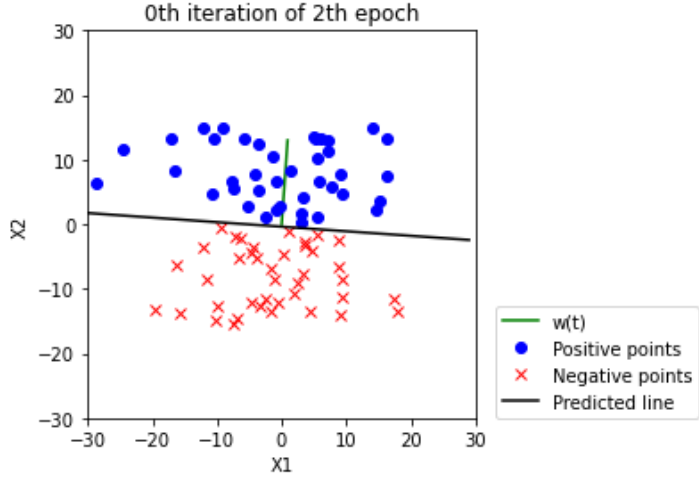
$$
\begin{aligned}
&\Rightarrow M\gamma \leq \sqrt{M} \\
&\Rightarrow M^2\gamma^2 \leq M \\
&\Rightarrow M \leq \frac{1}{\gamma^2}
\end{aligned}
$$

In practice, $\|\boldsymbol{w}^\star\|_2 = 1$ and $\|\boldsymbol{x}_i\|_2 \leq 1$ do not hold, but we can see the algorithm will converge in finite number of iterations.

A Python code "perceptron" is provided and you can alter parameters including training sample number and learning rate, and choose visualizing per iteration or epoch:

```
import numpy as np
from numpy import *
import matplotlib.pyplot as plt
import os
import sys
import math
import random


def generating_data(w,m,n):
    #w = [0, 0, 0.3]
    #m = 20
    #n = 20
```






```
The total epochs are: 2
```

Perceptron can perform AND operator. Training data are:

```
X=np.array([[1,1],[1,-1],[-1,1],[-1, -1]])
y=np.array([1,-1,-1,-1])
```

```
ppn = Perceptron(epochs=3, eta=0.1)

ppn.train(X, y)
print("The training output is ",y)
print("The network  output is ",ppn.predict(X))
```

```
Input is: 1 [1 1]
Network Output is  1  Teacher Output is  1
The old weight is  [0. 0. 0.]
The new weight i s [0. 0. 0.]
Input is: 1 [ 1 -1]
Network Output is  1  Teacher Output is  -1
The old weight is  [0. 0. 0.]
The new weight i s [-0.2 -0.2  0.2]
Input is: 1 [-1  1]
Network Output is  1  Teacher Output is  -1
The old weight is  [-0.2 -0.2  0.2]
The new weight i s [-0.4  0.   0. ]
Input is: 1 [-1 -1]
Network Output is  -1  Teacher Output is  -1
The old weight is  [-0.4  0.   0. ]
The new weight i s [-0.4  0.   0. ]
Input is: 1 [1 1]
Network Output is  -1  Teacher Output is  1
The old weight is  [-0.4  0.   0. ]
The new weight i s [-0.2  0.2  0.2]
Input is: 1 [ 1 -1]
Network Output is  -1  Teacher Output is  -1
The old weight is  [-0.2  0.2  0.2]
The new weight i s [-0.2  0.2  0.2]
Input is: 1 [-1  1]
Network Output is  -1  Teacher Output is  -1
The old weight is  [-0.2  0.2  0.2]
The new weight i s [-0.2  0.2  0.2]
Input is: 1 [-1 -1]
Network Output is  -1  Teacher Output is  -1
The old weight is  [-0.2  0.2  0.2]
The new weight i s [-0.2  0.2  0.2]
Input is: 1 [1 1]
Network Output is  1  Teacher Output is  1
The old weight is  [-0.2  0.2  0.2]
The new weight i s [-0.2  0.2  0.2]
Input is: 1 [ 1 -1]
Network Output is  -1  Teacher Output is  -1
The old weight is  [-0.2  0.2  0.2]
The new weight i s [-0.2  0.2  0.2]
Input is: 1 [-1  1]
Network Output is  -1  Teacher Output is  -1
The old weight is  [-0.2  0.2  0.2]
The new weight i s [-0.2  0.2  0.2]
Input is: 1 [-1 -1]
Network Output is  -1  Teacher Output is  -1
The old weight is  [-0.2  0.2  0.2]
The new weight i s [-0.2  0.2  0.2]
The training output is  [ 1 -1 -1 -1]
The network  output is  [ 1 -1 -1 -1]
```

Perceptron can classify flowers.

"Iris" a well-known dataset in the pattern recognition literature, which contains samples of 3 iris classes, Setosa, Versicolour and Virginica.

Source: IRIS Flower Prediction Using Machine Learning Algorithms (machinelearningsol.com)

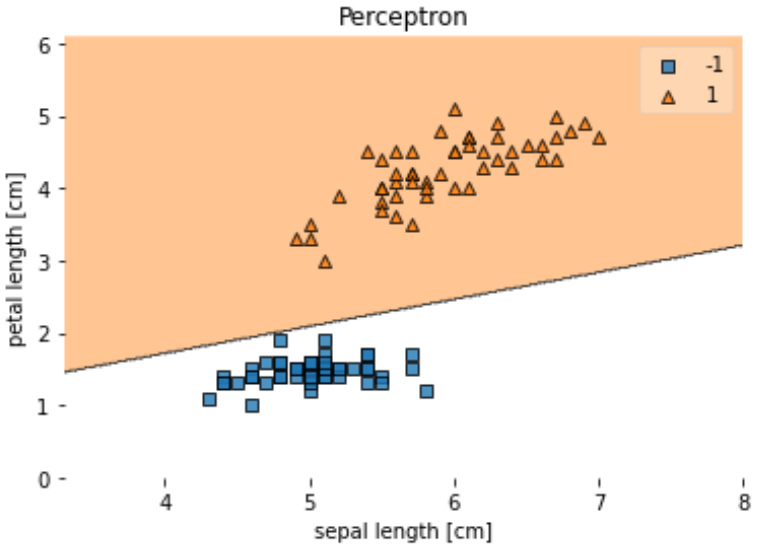
We just use two linearly separable classes Setosa and Versicolour, and two features, namely, sepal length and petal length, as input training data.

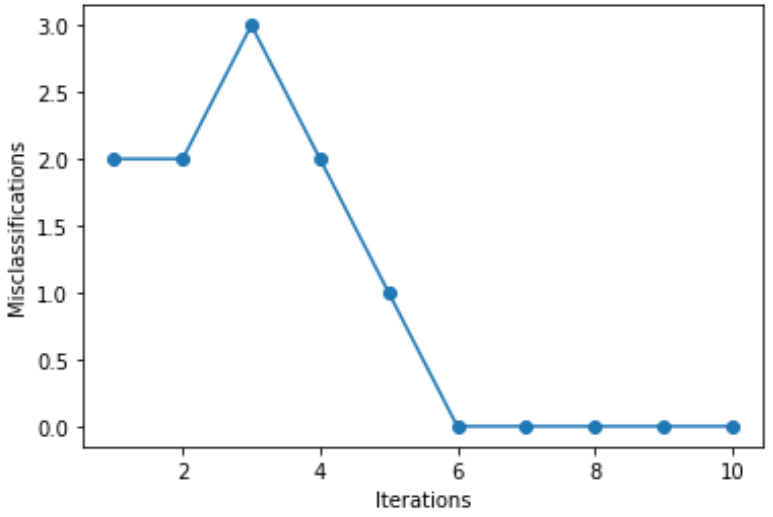
UCI Machine Learning Repository: Iris Data Set

In [2]:
```
import pandas as pd
df = pd.read_csv('iris.data', header=None)

# setosa and versicolor
y = df.iloc[0:100, 4].values
y = np.where(y == 'Iris-setosa', -1, 1)

# sepal length and petal length
X = df.iloc[0:100, [0,2]].values
```

In [3]:
```
%matplotlib inline
import matplotlib.pyplot as plt
from mlxtend.plotting import plot_decision_regions

ppn = Perceptron(epochs=10, eta=0.1)

ppn.train(X, y)
print('Weights: %s' % ppn.w_)
plot_decision_regions(X, y, clf=ppn)
plt.title('Perceptron')
plt.xlabel('sepal length [cm]')
plt.ylabel('petal length [cm]')
plt.show()

plt.plot(range(1, len(ppn.errors_)+1), ppn.errors_, marker='o')
plt.xlabel('Iterations')
plt.ylabel('Misclassifications')
plt.show()
```

```
Weights: [-0.4  -0.68  1.82]
```





For XOR, weights cannot converge.

```
Input is: 1 [1 1]
Network Output is  1  Teacher Output is  1
The old weight is  [0. 0. 0.]
The new weight i s [0. 0. 0.]
Input is: 1 [ 1 -1]
Network Output is  1  Teacher Output is  -1
The old weight is  [0. 0. 0.]
The new weight i s [-0.2 -0.2  0.2]
Input is: 1 [-1  1]
Network Output is  1  Teacher Output is  -1
The old weight is  [-0.2 -0.2  0.2]
The new weight i s [-0.4  0.   0. ]
Input is: 1 [-1 -1]
Network Output is  -1  Teacher Output is  1
The old weight is  [-0.4  0.   0. ]
The new weight i s [-0.2 -0.2 -0.2]
Input is: 1 [1 1]
Network Output is  -1  Teacher Output is  1
The old weight is  [-0.2 -0.2 -0.2]
The new weight i s [0. 0. 0.]
Input is: 1 [ 1 -1]
Network Output is  1  Teacher Output is  -1
The old weight is  [0. 0. 0.]
The new weight i s [-0.2 -0.2  0.2]
Input is: 1 [-1  1]
Network Output is  1  Teacher Output is  -1
The old weight is  [-0.2 -0.2  0.2]
The new weight i s [-0.4  0.   0. ]
Input is: 1 [-1 -1]
Network Output is  -1  Teacher Output is  1
The old weight is  [-0.4  0.   0. ]
The new weight i s [-0.2 -0.2 -0.2]
```

```
Input is: 1 [1 1]
Network Output is  -1  Teacher Output is  1
The old weight is  [-0.2 -0.2 -0.2]
The new weight i s [0. 0. 0.]
Input is: 1 [ 1 -1]
Network Output is  1  Teacher Output is  -1
The old weight is  [0. 0. 0.]
The new weight i s [-0.2 -0.2  0.2]
Input is: 1 [-1  1]
Network Output is  1  Teacher Output is  -1
The old weight is  [-0.2 -0.2  0.2]
The new weight i s [-0.4  0.   0. ]
Input is: 1 [-1 -1]
Network Output is  -1  Teacher Output is  1
The old weight is  [-0.4  0.   0. ]
The new weight i s [-0.2 -0.2 -0.2]
Input is: 1 [1 1]
Network Output is  -1  Teacher Output is  1
The old weight is  [-0.2 -0.2 -0.2]
The new weight i s [0. 0. 0.]
Input is: 1 [ 1 -1]
Network Output is  1  Teacher Output is  -1
The old weight is  [0. 0. 0.]
The new weight i s [-0.2 -0.2  0.2]
Input is: 1 [-1  1]
Network Output is  1  Teacher Output is  -1
The old weight is  [-0.2 -0.2  0.2]
The new weight i s [-0.4  0.   0. ]
Input is: 1 [-1 -1]
Network Output is  -1  Teacher Output is  1
The old weight is  [-0.4  0.   0. ]
The new weight i s [-0.2 -0.2 -0.2]
```

Batch Perceptron

Perceptron can be viewed as online learning as weight update is done each iteration using a single training input-output pair.

Batch perceptron performs weight update based on the entire training dataset at each iteration.

Its derivation is based on including all training data in a cost function $J(\boldsymbol{w})$ to be minimized:

$$J(\boldsymbol{w}) = \sum_{\boldsymbol{x}_i \in \mathcal{H}} -(\boldsymbol{w}^T \boldsymbol{x}_i) d_i = \sum_{i=1}^{N} \max(0, -(\boldsymbol{w}^T \boldsymbol{x}_i) d_i) \tag{10}$$

Here, $\mathcal{H}$ represents the set of misclassified samples. Recall that if $\boldsymbol{x}_i$ is misclassified, then $d_i(\boldsymbol{w}^T \boldsymbol{x}_i) < 0$. Note that $J(\boldsymbol{w})$ is not a linear function of $\boldsymbol{w}$ and gradient descent can be applied.

What is the minimum value of $J(\boldsymbol{w})$?

The gradient is simply:

$$\nabla(J(\boldsymbol{w})) = \left[\frac{\partial J(\boldsymbol{w})}{\partial w_0} \quad \frac{\partial J(\boldsymbol{w})}{\partial w_1} \quad \cdots \quad \frac{\partial J(\boldsymbol{w})}{\partial w_m}\right]^T = -\sum_{\boldsymbol{x}_i \in \mathcal{H}} \boldsymbol{x}_i d_i \quad (11)$$

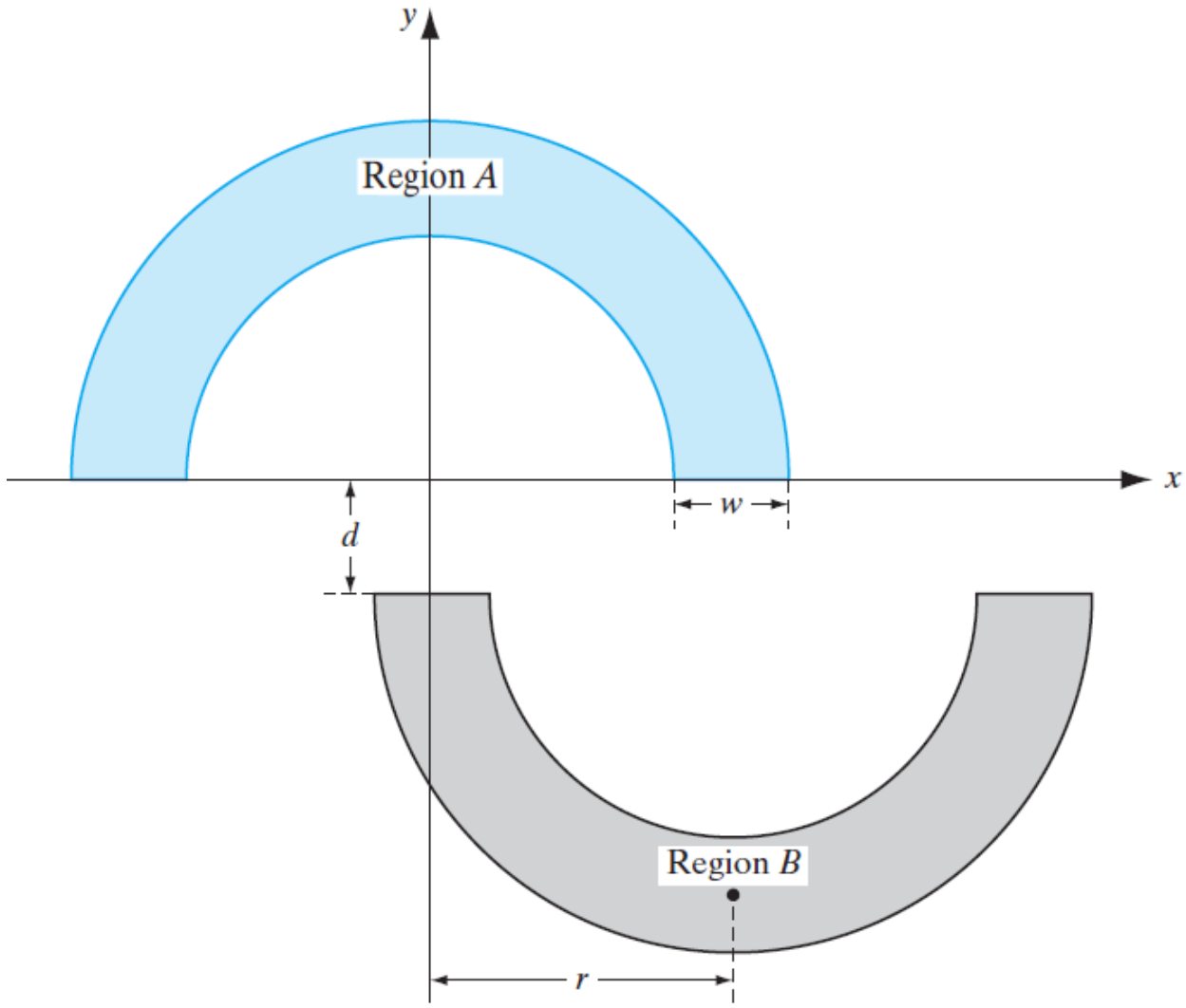
The gradient descent algorithm is then:

$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - \eta \nabla(J(\boldsymbol{w}^{(t)})) = \boldsymbol{w}^{(t)} + \eta \sum_{\boldsymbol{x}_i^{(t)} \in \mathcal{H}} \boldsymbol{x}_i^{(t)} d_i \quad (12)$$

Note that the misclassified samples $\boldsymbol{x}_i$ change during each iteration. Upon convergence, $\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)}$.

Under what condition convergence is guaranteed?

A MATLAB example for classifying data in 2 "moons" is also provided, where moon radius r, moon width w, separation between 2 moons d, number of training data, can be adjusted.

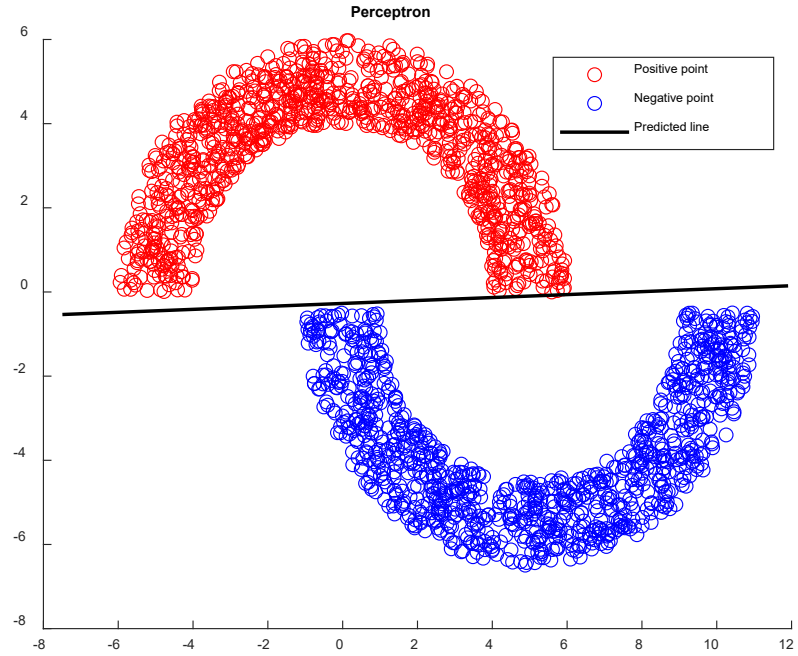


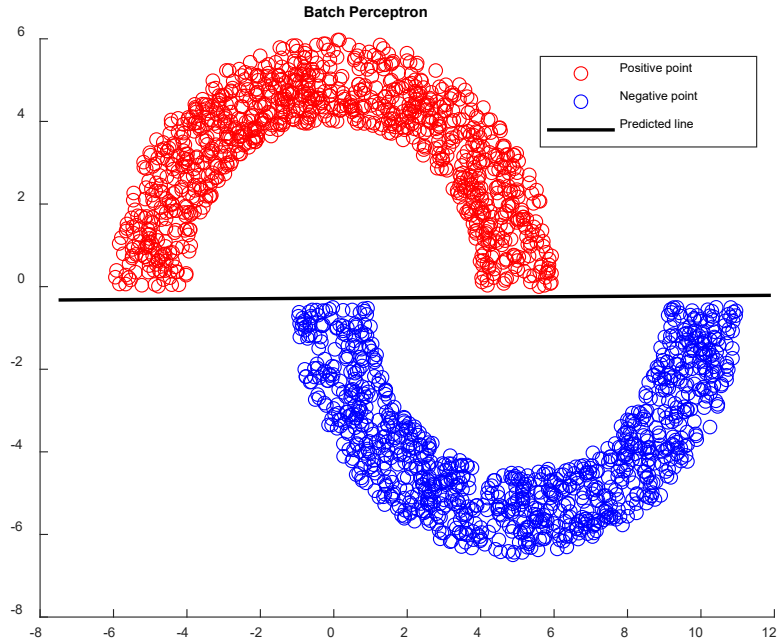
Try Demo.m

Demo.m

```
clear
close all
r = 5;
w = 2;
d = 0.5;
N = 1000;
seed = randn(1);
display = 0;
[X,Y,Xt] = generate_two_moons(r,w,d,N,seed,display);
stepSize = 1;
w0 = randn(3,1);
[w, n] = PLA(X,Y,w0,stepSize);
[Bw, Bn] = BatchPLA(X,Y,w0,stepSize);
figure(1)
show(X,N,w)
title('PLA')
figure(2)
show(X,N,Bw)
title('Batch PLA')
fprintf('Required epochs in PLA and Batch PLA are %d and %d, respectively.\n',n, Bn);
```

Required epochs for online and batch modes are 3 and 12.

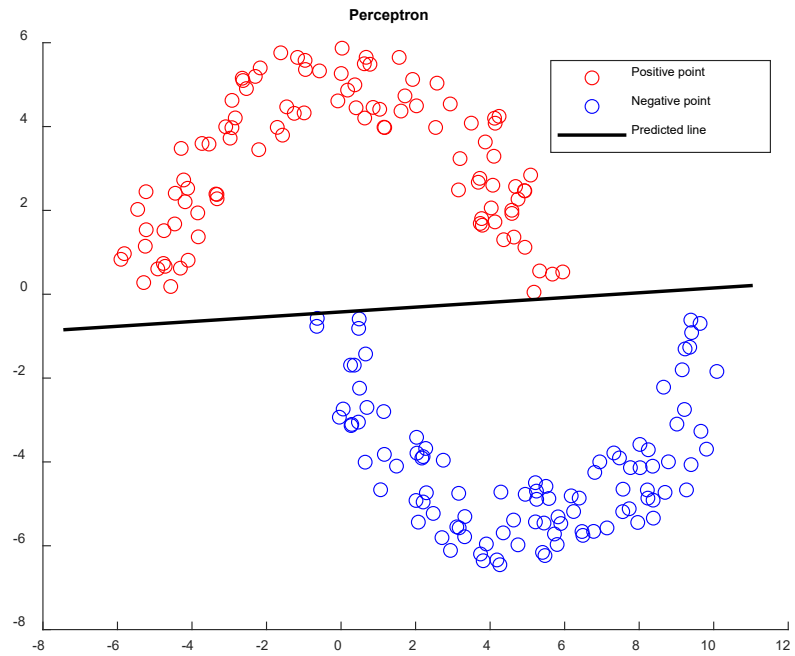


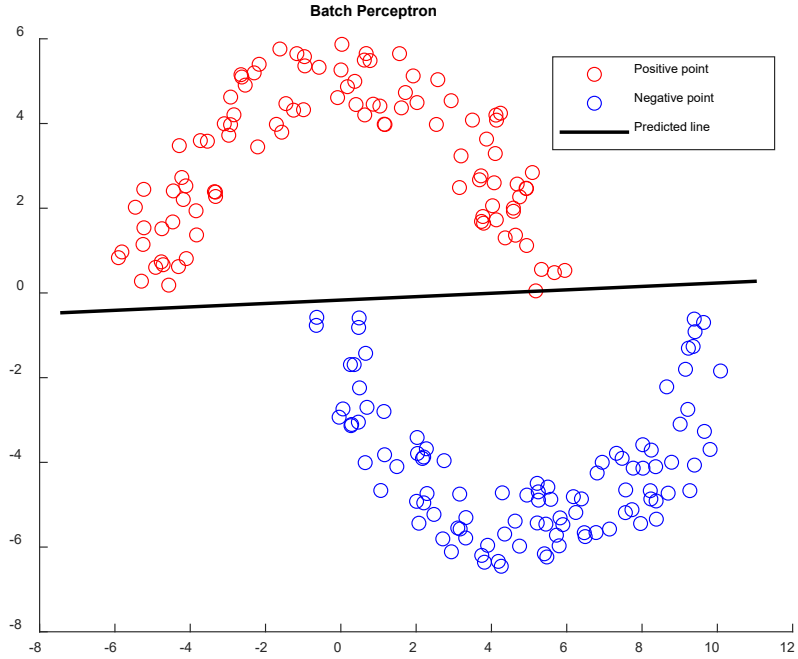


Note that the number of required epochs changes each time as the training data are randomly generated.

Try `N=100`:

Required epochs for online and batch modes are 6 and 5.

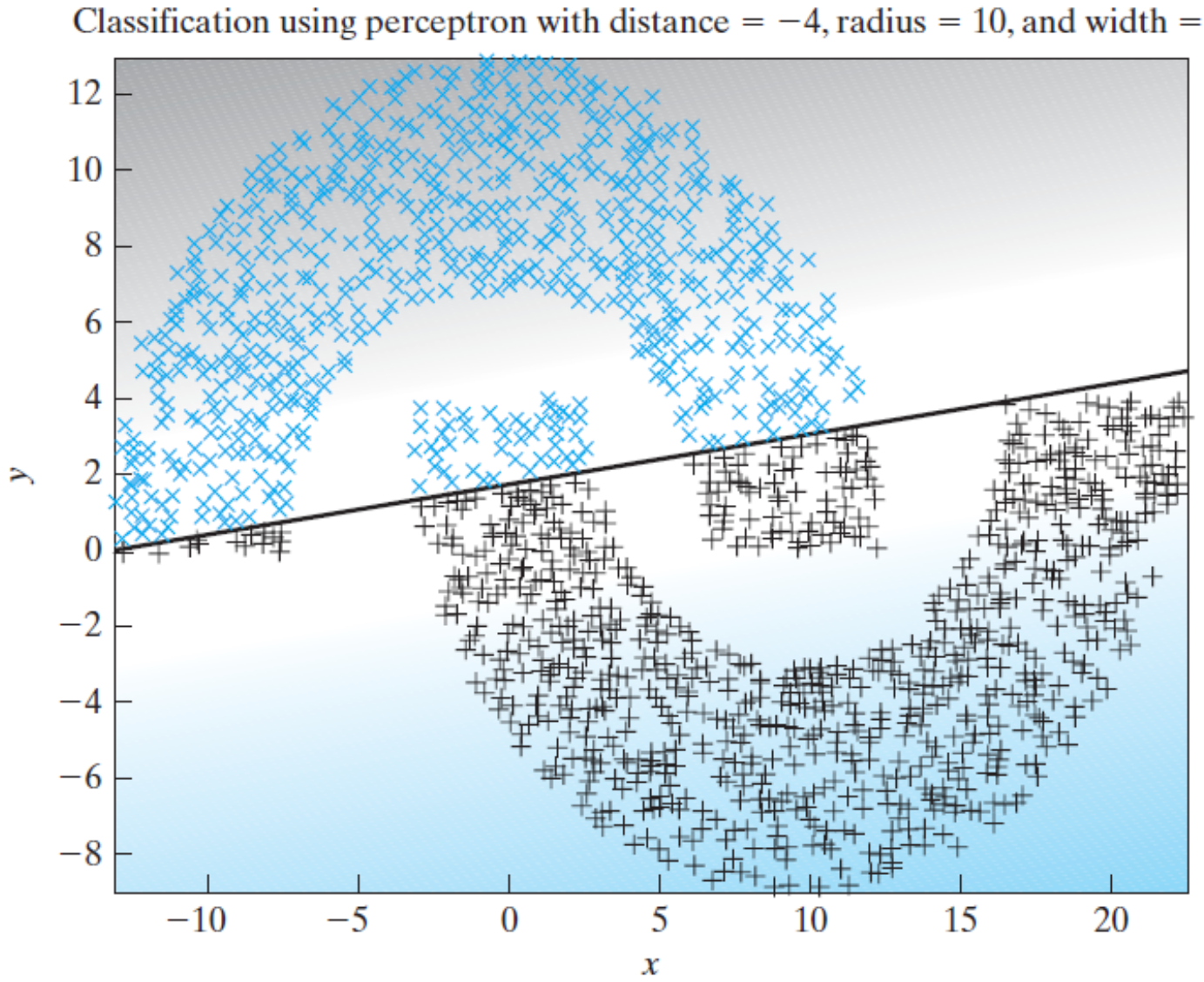




We see that both standard and batch perceptrons work properly although their decision boundaries are different.

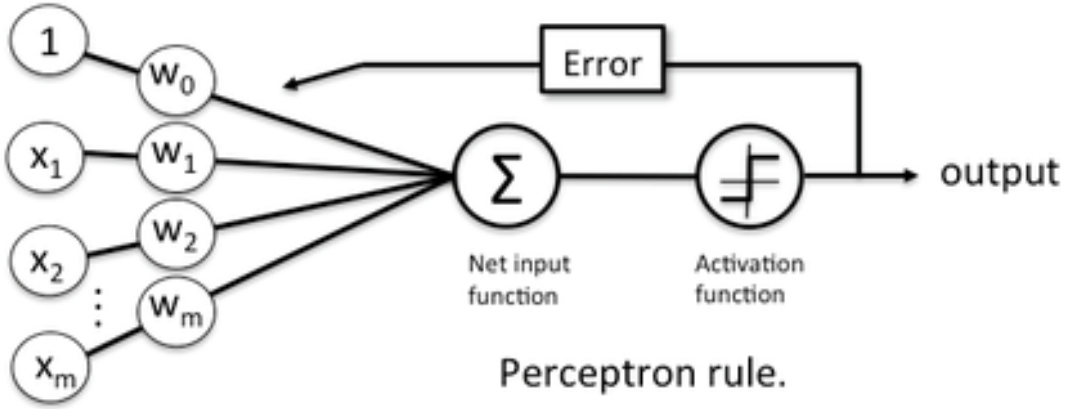
From the results, it is difficult to tell which scheme involves smaller number of iterations/epochs.

When $d < 0$, the 2 classes are not linearly separable. The weights will not converge, and the boundary is changing.



Linear Model for Classification

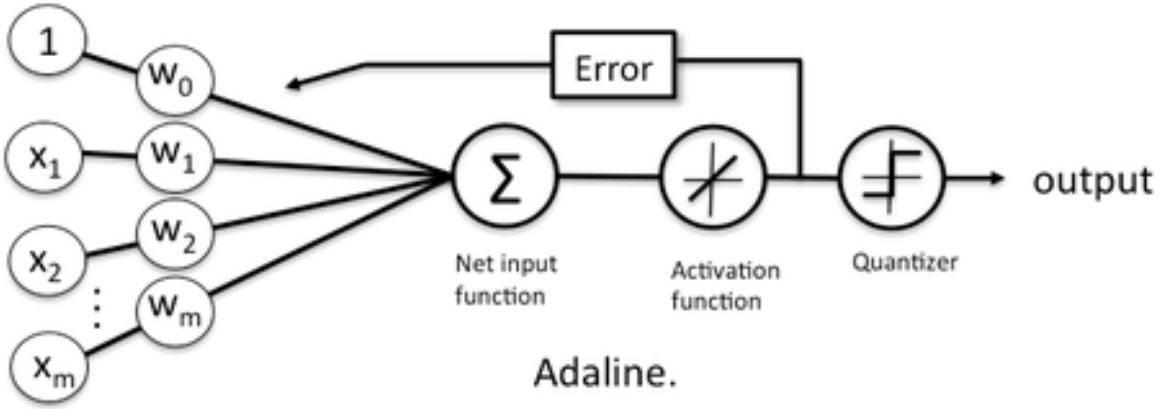
This approach basically corresponds to a linear neuron, and there are no nonlinear operations in weight update.



Perceptron rule.

From (5)-(7), error function is nonlinear:

$$d - \text{sign}(\boldsymbol{w}^T\boldsymbol{x}) = \{0, -2, 2\}$$



Adaline.

Error function is linear:

$$d - \boldsymbol{w}^T\boldsymbol{x}$$

Source: What is the difference between a Perceptron, Adaline, and neural network model? (sebastianraschka.com)

Following (4), we model the training input and output as:

$$d = \boldsymbol{w}^T \boldsymbol{x} + \epsilon \tag{13}$$

where

$$\boldsymbol{x} = [1 \; x_1 \; \cdots \; x_m]^T \quad \text{and} \quad \boldsymbol{w} = [w_0 \; w_1 \; \cdots \; w_m]^T$$

This is also called linear regression model where $[x_1 \; \cdots \; x_m]^T$ is known as regressor, while $\boldsymbol{w}$ is called parameter vector.

As $d - \boldsymbol{w}^T \boldsymbol{x}$ will not be exactly zero even for correct classification, ϵ accounts for this difference.

We first apply the probability viewpoints based on Bayes' rule to find $\boldsymbol{w}$.

The basic setup is that $\boldsymbol{x}$ (ignore the first element), $\boldsymbol{w}$ and ϵ are considered as random variables, with $p(\boldsymbol{w}, \boldsymbol{x}) = p(\boldsymbol{w})p(\boldsymbol{x})$, i.e., $\boldsymbol{w}$ and $\boldsymbol{x}$ are independent.

Batch Mode Solutions for Linear Model

We first investigate the joint probability distribution function (PDF) of $\boldsymbol{w}$ and d conditional on $\boldsymbol{x}$, denoted by $p(\boldsymbol{w}, d|\boldsymbol{x})$.

Applying Bayes' rule and independence of $\boldsymbol{x}$ and $\boldsymbol{w}$:

$$p(\boldsymbol{w}, d|\boldsymbol{x}) = p(\boldsymbol{w}|d, \boldsymbol{x}) \cdot p(d|\boldsymbol{x}) = p(d|\boldsymbol{w}, \boldsymbol{x}) \cdot p(\boldsymbol{w}|\boldsymbol{x}) = p(d|\boldsymbol{w}, \boldsymbol{x}) \cdot p(\boldsymbol{w})$$

We get:

$$p(\boldsymbol{w}|d, \boldsymbol{x}) = \frac{p(d|\boldsymbol{w}, \boldsymbol{x}) \cdot p(\boldsymbol{w})}{p(d|\boldsymbol{x})} \tag{14}$$

- $p(d|\boldsymbol{w}, \boldsymbol{x})$ is called observation density or likelihood function.
- $p(\boldsymbol{w}) = \pi(\boldsymbol{w})$ is called prior, i.e., it is our prior knowledge about $\boldsymbol{w}$ before observing $\{\boldsymbol{x}, d\}$.
- $p(\boldsymbol{w}|d, \boldsymbol{x})$ is called the posterior density is the conditional PDF after observing $\{\boldsymbol{x}, d\}$.
- $p(d|\boldsymbol{x})$ is called evidence.

The maximum likelihood (ML) estimate of $\boldsymbol{w}$ is:

$$\hat{\boldsymbol{w}}_{\text{ML}} = \arg\max_{\boldsymbol{w}} p(d|\boldsymbol{w}, \boldsymbol{x}) \tag{15}$$

The maximum *a posteriori* (MAP) estimate of $\boldsymbol{w}$ is:

$$\hat{\boldsymbol{w}}_{\text{MAP}} = \arg\max_{\boldsymbol{w}} p(\boldsymbol{w}|d, \boldsymbol{x}) \tag{16}$$

We also see that:

$$p(\boldsymbol{w}|d, \boldsymbol{x}) \propto p(d|\boldsymbol{w}, \boldsymbol{x}) \cdot p(\boldsymbol{w}) \tag{17}$$

and $p(d|\boldsymbol{x})$ is not required in the calculation.

Suppose there are N pairs of training samples $\{\boldsymbol{x}_i, d_i\}_{i=1}^{N}$:

$$d_i = \boldsymbol{w}^T \boldsymbol{x}_i + \epsilon_i, \quad , i = 1, \cdots, N \tag{18}$$

A zero-mean Gaussian environment is considered with:

Assumption 1

Independence and identical distribution: The N training data are independent and identically distributed (IID).

Assumption 2

Gaussianity: All $\{\epsilon_i\}_{i=1}^N$ are IID zero-mean Gaussian distributed with variance σ^2, i.e., $\epsilon_i \sim \mathcal{N}(0, \sigma^2)$, and the PDF is:

$$p(\epsilon_i) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2\sigma^2}\epsilon_i^2}, \quad i = 1, \cdots, N \tag{19}$$

Assumption 3

Stationarity: Among this set of $\{\boldsymbol{x}_i, d_i\}_{i=1}^N$, $\boldsymbol{w}$ is fixed. We further assume that all elements in $\boldsymbol{w}$ are IID and zero-mean Gaussian distributed, i.e., $w_k \sim \mathcal{N}(0, \sigma_w^2)$:

$$\pi(w_k) = p(w_k) = \frac{1}{\sqrt{2\pi\sigma_w^2}} e^{-\frac{1}{2\sigma_w^2}w_k^2}, \quad k = 0, \cdots, m \tag{20}$$

For a given $\boldsymbol{x}_i$ and fixed $\boldsymbol{w}$, $p(d|\boldsymbol{w},\boldsymbol{x})$ is characterized by $\epsilon_i \sim \mathcal{N}(0,\sigma^2)$:

$$p(d_i|\boldsymbol{w},\boldsymbol{x}_i) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2\sigma^2}(d_i - \boldsymbol{w}^T\boldsymbol{x}_i)^2}, \quad i = 1, \cdots, N \tag{21}$$

Together with the IID Assumption 1, we obtain:

$$\begin{aligned} p(\{d_i\}|\boldsymbol{w},\{\boldsymbol{x}_i\}) &= \prod_{i=1}^{N} p(d_i|\boldsymbol{w},\boldsymbol{x}_i) = \frac{1}{\left(2\pi\sigma^2\right)^{N/2}} \prod_{i=1}^{N} e^{-\frac{1}{2\sigma^2}(d_i - \boldsymbol{w}^T\boldsymbol{x}_i)^2} \\ &= \frac{1}{\left(2\pi\sigma^2\right)^{N/2}} e^{-\frac{1}{2\sigma^2}\sum_{i=1}^{N}(d_i - \boldsymbol{w}^T\boldsymbol{x}_i)^2} \end{aligned} \tag{22}$$

Similarly, from Assumption 3:

$$\pi(\boldsymbol{w}) = \prod_{k=0}^{m} \pi(w_k) = \frac{1}{\left(2\pi\sigma_w^2\right)^{N/2}} e^{-\frac{1}{2\sigma_w^2}\sum_{k=0}^{m} w_k^2} = \frac{1}{\left(2\pi\sigma_w^2\right)^{N/2}} e^{-\frac{1}{2\sigma_w^2}\boldsymbol{w}^T\boldsymbol{w}} \tag{23}$$

To obtain the ML solution, we see that maximizing (22) is equivalent to minimizing the least squares (LS):

$$J_{\mathrm{ML}}(\boldsymbol{w}) = \sum_{i=1}^{N}(d_i - \boldsymbol{w}^T\boldsymbol{x}_i)^2 = \sum_{i=1}^{N}(d_i - \boldsymbol{x}_i^T\boldsymbol{w})^2$$

which can be compactly expressed as:

$$J_{\mathrm{ML}}(\boldsymbol{w}) = \|\boldsymbol{d} - \boldsymbol{X}\boldsymbol{w}\|_2^2 = (\boldsymbol{d} - \boldsymbol{X}\boldsymbol{w})^T(\boldsymbol{d} - \boldsymbol{X}\boldsymbol{w}) = J_{\mathrm{LS}}(\boldsymbol{w}) \quad (24)$$

by defining

$$\boldsymbol{d} = \begin{bmatrix} d_1 \\ \vdots \\ d_N \end{bmatrix} \quad \text{and} \quad \boldsymbol{X} = \begin{bmatrix} \boldsymbol{x}_1^T \\ \vdots \\ \boldsymbol{x}_N^T \end{bmatrix}$$

That is, the ML solution is the same as the LS solution:

$$\hat{\boldsymbol{w}}_{\mathrm{ML}} = \arg\max_{\boldsymbol{w}} p(d|\boldsymbol{w}, \boldsymbol{x}) = \hat{\boldsymbol{w}}_{\mathrm{LS}} = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{d} \quad (25)$$

Is the ML solution always equal to the LS solution?

According to (17), the MAP solution maximizes:

$$J_{\text{MAP}}(\boldsymbol{w}) = \frac{1}{(2\pi\sigma^2)^{N/2}} e^{-\frac{1}{2\sigma^2}\|\boldsymbol{d}-\boldsymbol{X}\boldsymbol{w}\|_2^2} \cdot \frac{1}{(2\pi\sigma_w^2)^{N/2}} e^{-\frac{1}{2\sigma_w^2}\boldsymbol{w}^T\boldsymbol{w}}$$

which is equivalent to minimizing regularized LS (RLS):

$$J_{\text{MAP}}(\boldsymbol{w}) = J_{\text{RLS}}(\boldsymbol{w}) = \frac{1}{2}\|\boldsymbol{d}-\boldsymbol{X}\boldsymbol{w}\|_2^2 + \frac{\lambda}{2}\boldsymbol{w}^T\boldsymbol{w}, \quad \lambda = \frac{\sigma^2}{\sigma_w^2} \tag{26}$$

where λ is known as regularization parameter. Differentiating (26) w.r.t. $\boldsymbol{w}$ and then setting the resultant expression to $\mathbf{0}$, we get:

$$-\boldsymbol{X}^T\boldsymbol{d} + \boldsymbol{X}^T\boldsymbol{X}\hat{\boldsymbol{w}}_{\text{MAP}} + \lambda\hat{\boldsymbol{w}}_{\text{MAP}} = \mathbf{0} \Rightarrow \left(\boldsymbol{X}^T\boldsymbol{X} + \lambda\boldsymbol{I}_N\right)\hat{\boldsymbol{w}}_{\text{MAP}} = \boldsymbol{X}^T\boldsymbol{d}$$

$$\Rightarrow \hat{\boldsymbol{w}}_{\text{MAP}} = \left(\boldsymbol{X}^T\boldsymbol{X} + \lambda\boldsymbol{I}_N\right)^{-1}\boldsymbol{X}^T\boldsymbol{d} \tag{27}$$

Which solution is better? ML or MAP? Why?

Consider a limiting case of $w_k \sim \mathcal{N}(0, \sigma_w^2)$ such that $\sigma_w^2 \to \infty$, i.e., we have no prior information about $\boldsymbol{w}$.

This results in $\lambda = 0$ which is the same as the ML or LS solution.

On the other hand, the smaller the σ_w^2, the larger the λ, indicating that the second component $\boldsymbol{w}^T\boldsymbol{w}$ increases its importance in the RLS cost function.

Since we may not have the prior information of $\boldsymbol{w}$ and even the probability distributions of the training data, only the performance of the LS solution $\hat{\boldsymbol{w}}_{\text{LS}} = \boldsymbol{w}^\star$ is examined further.

Even 2 iris classes are not linearly separable, LS provides a reasonable boundary:

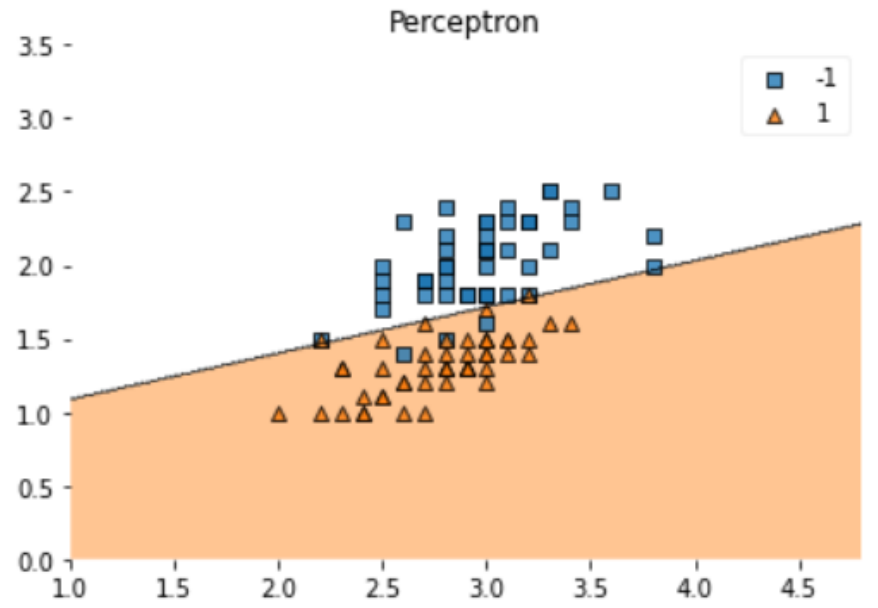
LS_Classify_Flowers (autosaved)

View Insert Cell Kernel Widgets Help

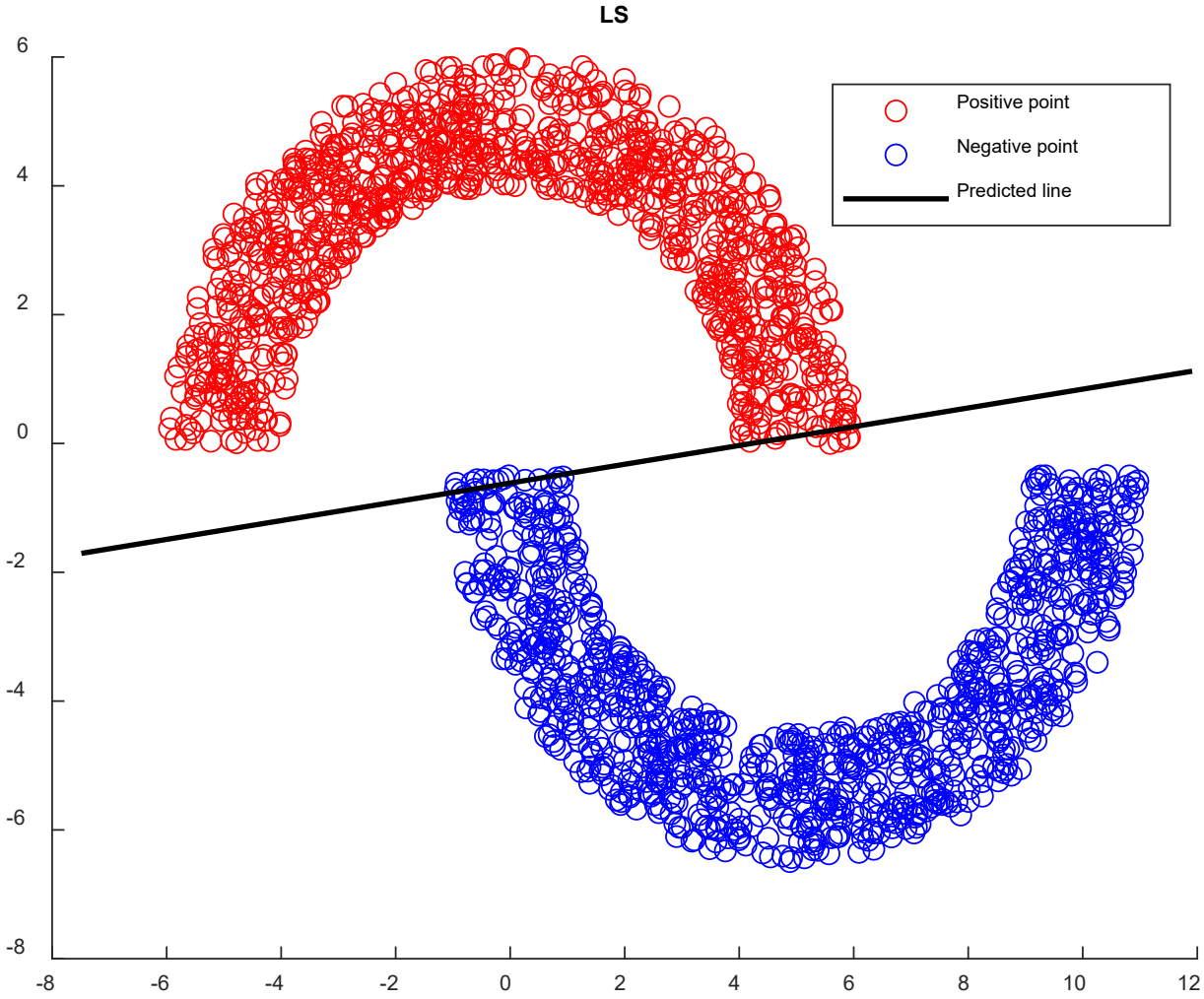
Run Code

```
import numpy as np

class Perceptron(object):


    def train(self, X, y):


        kk=(y.shape)
        N=kk[0]
        addx=np.ones((N,1))
        XX2=np.concatenate((addx,X),axis=1)
        yy=np.transpose(XX2).dot(y)
        XXXX=np.linalg.inv(np.transpose(XX2).dot(XX2))
        self.w_=np.transpose(XXXX).dot(yy)
        return self

    def net_input(self, X):
        return np.dot(X, self.w_[1:]) + self.w_[0]

    def predict(self, X):
        return np.where(self.net_input(X) >= 0.0, 1, -1)
```

```
Weights: [ 1.76226811  0.71488925 -2.27650957]
```



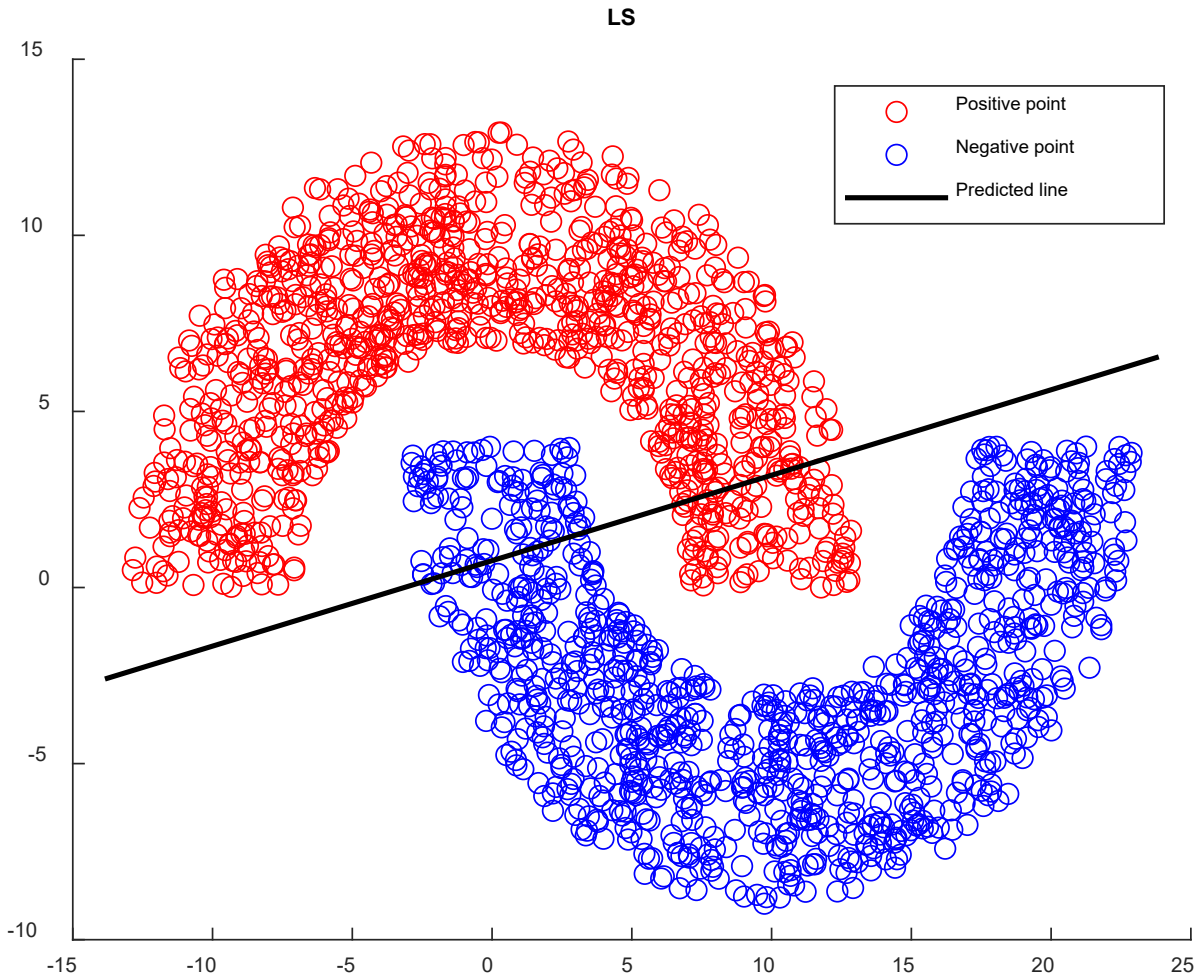
```
>>LM_w = LM(X,Y);
```



Demo.m

```
clear
close all
r = 5;
w = 2;
d = 0.5;
N = 1000;
```

```
r = 10;
w = 6;
d = -4;
N = 1000;
```



Comparing the perceptron and LS solution, we may conclude:

- Both algorithms construct linear and distinct decision boundaries.
- Even for the linearly separable scenario, the LS solution cannot achieve classification error of 0 while the perceptron approach will provide perfect classification in this case.
- The LS solution provides the solution in one step for both linearly and non-linearly separable scenarios, and thus there is no convergence problem as in the perceptron applied in the non-linearly separable case.

Iterative Solutions for Linear Model

Minimization of the ML/LS cost function can also be achieved using iterative techniques such as Newton's method and gradient descent:

$$\boldsymbol{w}^{(k+1)} = \boldsymbol{w}^{(k)} - \boldsymbol{H}^{-1}(J(\boldsymbol{w}^{(k)}))\boldsymbol{\nabla}(J(\boldsymbol{w}^{(k)})) \tag{28}$$

and

$$\boldsymbol{w}^{(k+1)} = \boldsymbol{w}^{(k)} - 0.5\mu\boldsymbol{\nabla}(J(\boldsymbol{w}^{(k)})) \tag{29}$$

The gradient vector of (24) is:

$$\boldsymbol{\nabla}(J(\boldsymbol{w}^{(k)})) = -2\boldsymbol{X}^T\boldsymbol{d} + 2\boldsymbol{X}^T\boldsymbol{X}\boldsymbol{w}^{(k)} = -2\boldsymbol{X}^T\left(\boldsymbol{d} - \boldsymbol{X}\boldsymbol{w}^{(k)}\right) \tag{30}$$

Differentiating (30) once w.r.t. $\boldsymbol{w}^{(k)}$ yields the Hessian matrix:

$$\boldsymbol{H}(J(\boldsymbol{w}^{(k)})) = 2\boldsymbol{X}^T\boldsymbol{X} \tag{31}$$

Using (30)-(31), the adjustment term in (28) is:

$$\begin{aligned}\boldsymbol{H}^{-1}(J(\boldsymbol{w}^{(k)}))\boldsymbol{\nabla}(J(\boldsymbol{w}^{(k)})) &= (2\boldsymbol{X}^T\boldsymbol{X})^{-1}\cdot -2\boldsymbol{X}^T\left(\boldsymbol{d}-\boldsymbol{X}\boldsymbol{w}^{(k)}\right)\\ &= -(\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{d}+(\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{X}\boldsymbol{w}^{(k)}\\ &= -\boldsymbol{w}^{\star}+\boldsymbol{w}^{(k)}\end{aligned}$$

To obtain better insight and ease the convergence proof, we introduce a step size μ in (28):

$$\boldsymbol{w}^{(k+1)}=\boldsymbol{w}^{(k)}-\mu\boldsymbol{H}^{-1}(J(\boldsymbol{w}^{(k)}))\boldsymbol{\nabla}(J(\boldsymbol{w}^{(k)}))=\boldsymbol{w}^{(k)}+\mu(\boldsymbol{w}^{\star}-\boldsymbol{w}^{(k)})$$

or

$$\boldsymbol{w}^{(k+1)}=(1-\mu)\boldsymbol{w}^{(k)}+\mu\boldsymbol{w}^{\star} \tag{32}$$

$$(1-\mu)\cdot\{\boldsymbol{w}^{(k)}=(1-\mu)\boldsymbol{w}^{(k-1)}+\mu\boldsymbol{w}^{\star}\}$$

$$(1-\mu)^2\cdot\{\boldsymbol{w}^{(k-1)}=(1-\mu)\boldsymbol{w}^{(k-2)}+\mu\boldsymbol{w}^{\star}\}$$

$$\ldots$$

$$(1-\mu)^k\cdot\{\boldsymbol{w}^{(1)}=(1-\mu)\boldsymbol{w}^{(0)}+\mu\boldsymbol{w}^{\star}\}$$

Adding all these $k+1$ equations yields:

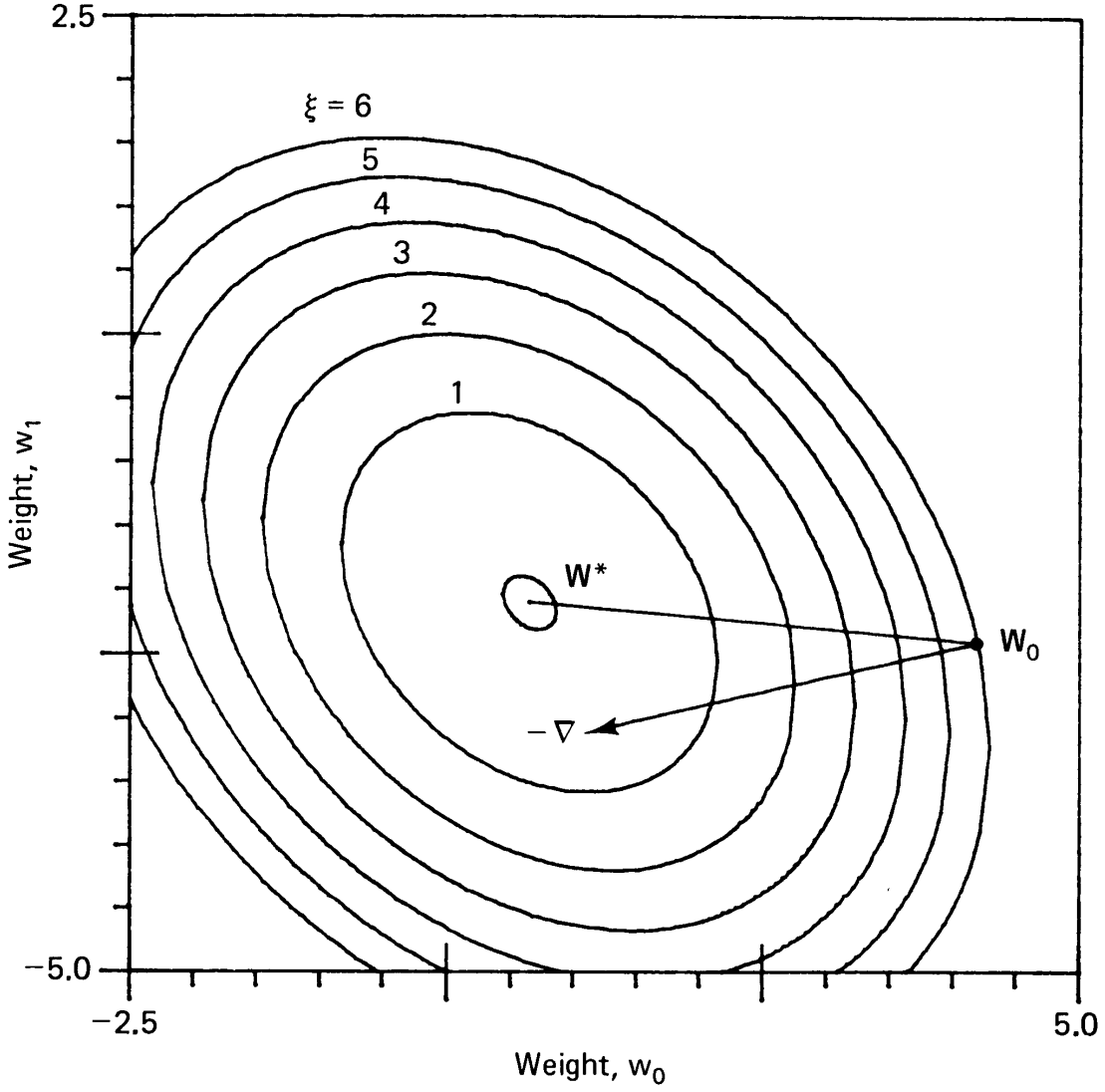
$$\begin{aligned}
\boldsymbol{w}^{(k+1)} &= (1-\mu)^{k+1}\boldsymbol{w}^{(0)} + \mu\boldsymbol{w}^{\star}(1+(1-\mu)+\cdots+(1-\mu)^{k}) \\
&= (1-\mu)^{k+1}\boldsymbol{w}^{(0)} + \mu\boldsymbol{w}^{\star}\left[\frac{1-(1-\mu)^{k+1}}{1-(1-\mu)}\right] \\
&= (1-\mu)^{k+1}\boldsymbol{w}^{(0)} + \boldsymbol{w}^{\star}\left[1-(1-\mu)^{k+1}\right] \\
&= \boldsymbol{w}^{\star} + (1-\mu)^{k+1}\left[\boldsymbol{w}^{(0)} - \boldsymbol{w}^{\star}\right]
\end{aligned}$$

Hence we easily see that when $k \to \infty$, $\boldsymbol{w}^{(k)} = \boldsymbol{w}^{\star}$ if

$$\left|1-\mu\right| < 1 \Rightarrow 0 < \mu < 2$$

In particular, when $\mu = 1$, corresponding to the Newton's method, the algorithm converges in one step for any choices of $\boldsymbol{w}^{(0)}$ in theory.

An example of the contour of $J_{\mathrm{LS}}(\boldsymbol{w})$ using the Newton's method with $\mu = 1$ and 2 weights is illustrated below.



On the other hand, using (30), (29) becomes:

$$\boldsymbol{w}^{(k+1)} = \boldsymbol{w}^{(k)} + \mu \boldsymbol{X}^T \left(\boldsymbol{d} - \boldsymbol{X} \boldsymbol{w}^{(k)} \right) \tag{33}$$

To study convergence, we rewrite (33) as:

$$\begin{aligned} \boldsymbol{w}^{(k+1)} &= [\boldsymbol{I}_{m+1} - \mu \boldsymbol{X}^T \boldsymbol{X}] \boldsymbol{w}^{(k)} + \mu \boldsymbol{X}^T \boldsymbol{d} \\ &= [\boldsymbol{I}_{m+1} - \mu \boldsymbol{X}^T \boldsymbol{X}] (\boldsymbol{w}^{(k)} - \boldsymbol{w}^\star) + \mu \boldsymbol{X}^T \boldsymbol{d} + [\boldsymbol{I}_{m+1} - \mu \boldsymbol{X}^T \boldsymbol{X}] \boldsymbol{w}^\star \\ &= [\boldsymbol{I}_{m+1} - \mu \boldsymbol{X}^T \boldsymbol{X}] (\boldsymbol{w}^{(k)} - \boldsymbol{w}^\star) + \boldsymbol{w}^\star + \mu [\boldsymbol{X}^T \boldsymbol{d} - \boldsymbol{X}^T \boldsymbol{X} \boldsymbol{w}^\star] \\ &= [\boldsymbol{I}_{m+1} - \mu \boldsymbol{X}^T \boldsymbol{X}] (\boldsymbol{w}^{(k)} - \boldsymbol{w}^\star) + \boldsymbol{w}^\star \end{aligned}$$

Let $\boldsymbol{v}^{(k)} = \boldsymbol{w}^{(k)} - \boldsymbol{w}^\star$. Then:

$$\boldsymbol{v}^{(k+1)} = [\boldsymbol{I}_{m+1} - \mu \boldsymbol{X}^T \boldsymbol{X}] \boldsymbol{v}^{(k)} \tag{34}$$

Applying eigenvalue decomposition (EVD) on $\boldsymbol{X}^T \boldsymbol{X}$ yields

$$\boldsymbol{X}^T \boldsymbol{X} = \boldsymbol{Q} \boldsymbol{\Lambda} \boldsymbol{Q}^{-1} = \boldsymbol{Q} \boldsymbol{\Lambda} \boldsymbol{Q}^T \tag{35}$$

As $\boldsymbol{X}^T\boldsymbol{X}$ is symmetric, the EVD has 3 additional properties:

$$\boldsymbol{Q}^{-1} = \boldsymbol{Q}^T$$

$$\boldsymbol{Q}^T\boldsymbol{Q} = \boldsymbol{Q}\boldsymbol{Q}^T = \boldsymbol{I}_{m+1}$$

$$\boldsymbol{\Lambda} = \begin{bmatrix} \lambda_0 & & \\ & \ddots & \\ & & \lambda_m \end{bmatrix}, \quad \lambda_i > 0,\ i = 0, \cdots, m$$

Utilizing the results and let $\boldsymbol{v}^{(k)} = \boldsymbol{Q}\boldsymbol{u}^{(k)}$, we obtain:

$$\boldsymbol{v}^{(k+1)} = [\boldsymbol{I}_{m+1} - \mu\boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^T]\boldsymbol{v}^{(k)} \Rightarrow \boldsymbol{Q}\boldsymbol{u}^{(k+1)} = [\boldsymbol{I}_{m+1} - \mu\boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^T]\boldsymbol{Q}\boldsymbol{u}^{(k)}$$

$$\begin{aligned} \Rightarrow \boldsymbol{u}^{(k+1)} &= \boldsymbol{Q}^{-1}[\boldsymbol{I}_{m+1} - \mu\boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^T]\boldsymbol{Q}\boldsymbol{u}^{(k)} \\ &= [\boldsymbol{Q}^{-1}\boldsymbol{I}_{m+1}\boldsymbol{Q} - \mu\boldsymbol{Q}^{-1}\mu\boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^T\boldsymbol{Q}\boldsymbol{u}^{(k)} \\ &= [\boldsymbol{I}_{m+1} - \mu\boldsymbol{\Lambda}]\boldsymbol{u}^{(k)} \end{aligned}$$

The solution is:

$$\boldsymbol{u}^{(k)} = [\boldsymbol{I}_{m+1} - \mu\boldsymbol{\Lambda}]^k \boldsymbol{u}^{(0)}$$

We expect:

$$\lim_{k\to\infty} \boldsymbol{u}^{(k)} = \mathbf{0} \Rightarrow \lim_{k\to\infty} \boldsymbol{Q}^{-1}\boldsymbol{v}^{(k)} = \mathbf{0} \Rightarrow \lim_{k\to\infty} \boldsymbol{Q}^{-1}(\boldsymbol{w}^{(k)} - \boldsymbol{w}^\star) = \mathbf{0}$$

$$\Rightarrow \lim_{k\to\infty} \boldsymbol{w}^{(k)} = \boldsymbol{w}^\star$$

The convergence requirement is then:

$$\lim_{k\to\infty} [\boldsymbol{I}_{m+1} - \mu\boldsymbol{\Lambda}]^k = \begin{bmatrix} \lim\limits_{k\to\infty} (1-\mu\lambda_0)^k & & \\ & \cdots & \\ & & \lim\limits_{k\to\infty} (1-\mu\lambda_m)^k \end{bmatrix} = \mathbf{0}$$

or

$$|1 - \mu\lambda_i| < 1 \Rightarrow 0 < \mu < \frac{2}{\lambda_i}, \quad i = 0, \cdots, m$$

Denote the maximum eigenvalue as $\lambda_{\max}$, the algorithm converges if

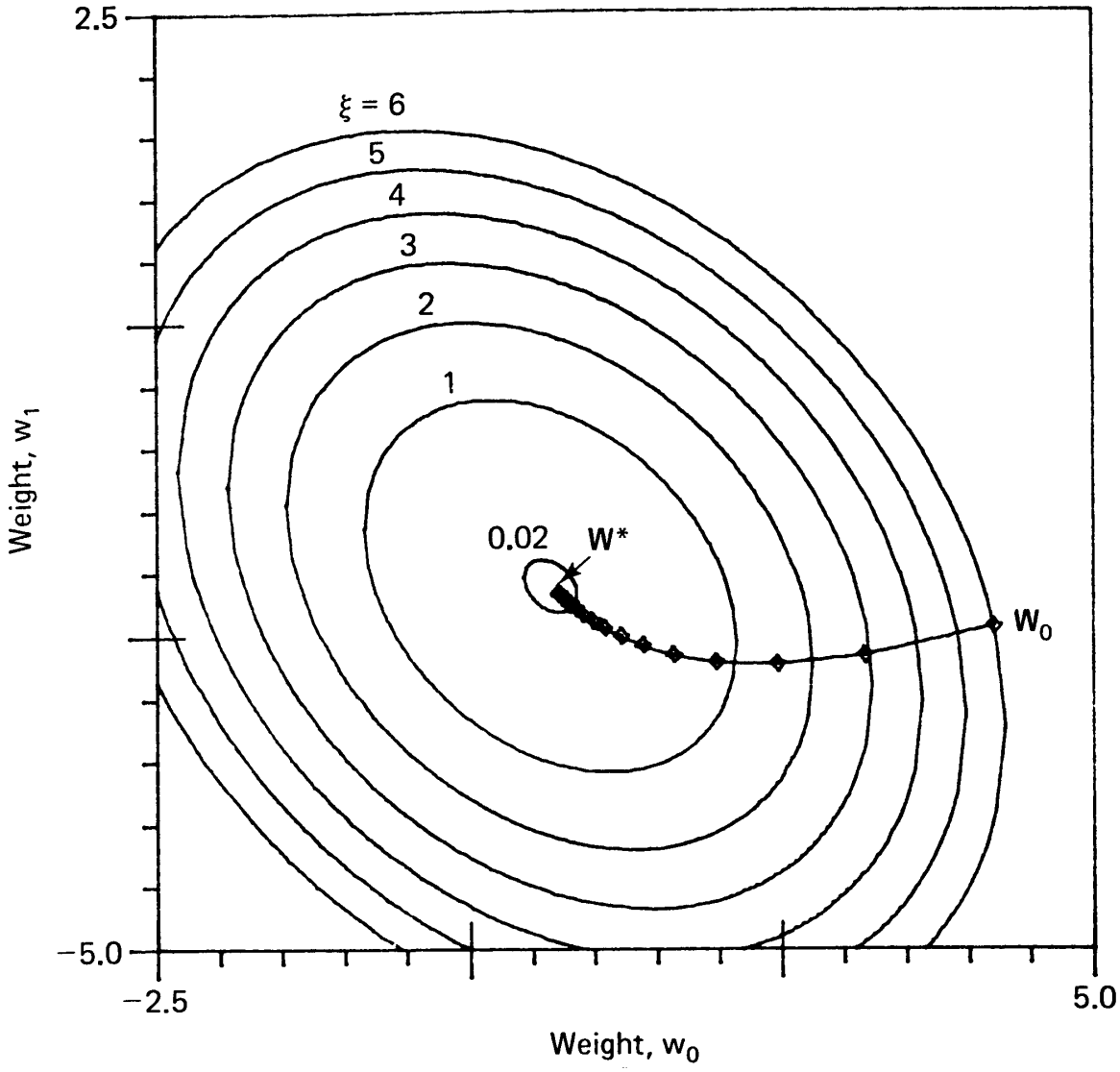
$$0 < \mu < \frac{2}{\lambda_{\max}}$$

That is, the convergence condition is governed by the largest eigenvalue of $\boldsymbol{X}^T\boldsymbol{X}$.

Since $\lambda_{\max}$ may not be available, in practice a sufficiently small value of μ is chosen, and the steepest descent algorithm converges in multiple steps for any choices of $\boldsymbol{w}^{(0)}$.

It is clear that for a larger μ, $(1-\mu\lambda_i)^k$ approaches 0 faster, indicating faster convergence. Nevertheless, the overall convergence is hindered by $(1-\mu\lambda_{\min})^k$ where $\lambda_{\min}$ denotes the minimum eigenvalue.

An example of the contour of $J_{\mathrm{LS}}(\boldsymbol{w})$ using the steepest descent with 2 weights is illustrated below.



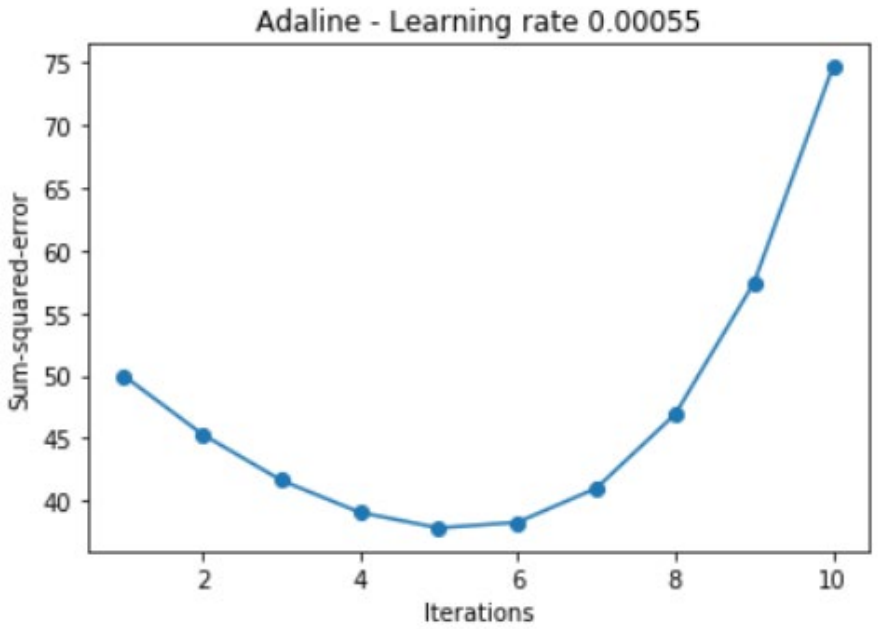
We examine the sum of squared error $\|\boldsymbol{d} - \boldsymbol{X}\boldsymbol{w}^{(k)}\|_2^2$ of (24) versus number of iterations for different values of μ.

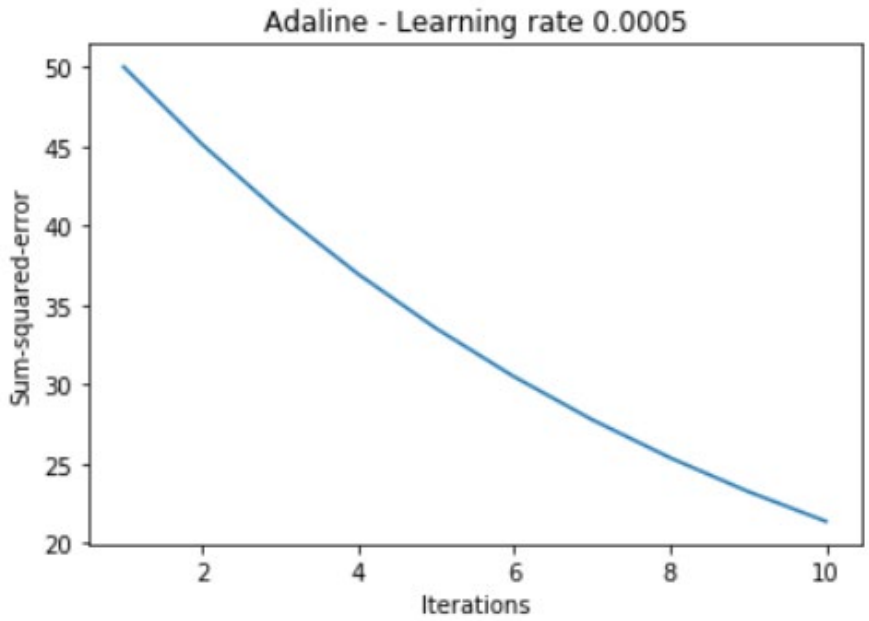
In [3]:
```
ada = AdalineGD(epochs=10, eta=0.00055).train(X, y)
plt.plot(range(1, len(ada.cost_)+1), ada.cost_, marker='o')
plt.xlabel('Iterations')
plt.ylabel('Sum-squared-error')
plt.title('Adaline - Learning rate 0.00055')
plt.show()
plot_decision_regions(X, y, clf=ada)
plt.title('Adaline - Gradient Descent')
plt.show()
```

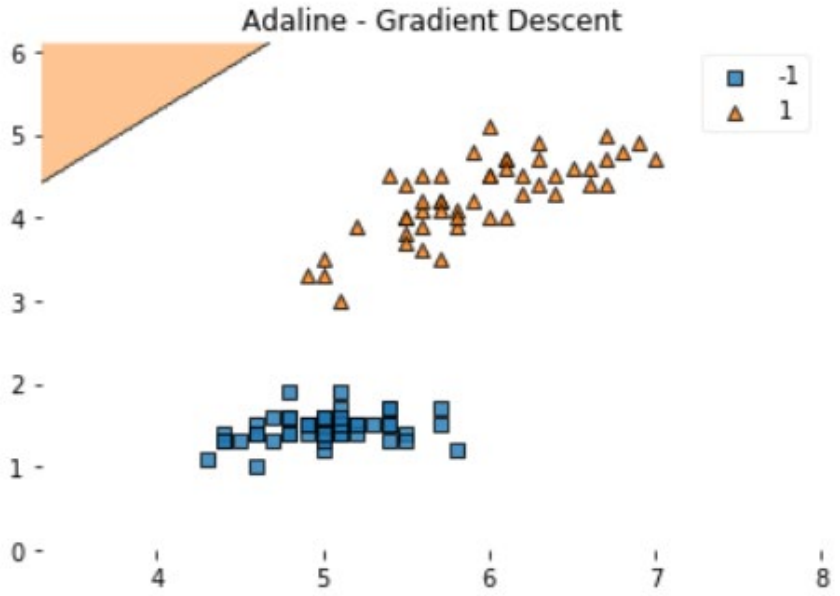
In [4]:
```
print(ada.w_)
```

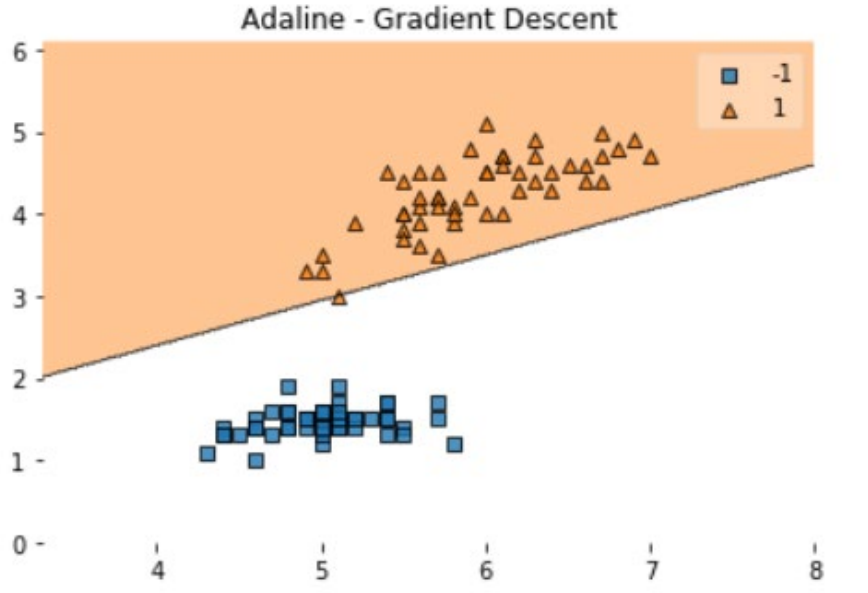
```
[-0.09824251 -0.34981954  0.28416519]
```

In [5]:
```
ada = AdalineGD(epochs=10, eta=0.0005).train(X, y)
plt.plot(range(1, len(ada.cost_)+1), ada.cost_, marker=' ')
plt.xlabel('Iterations')
plt.ylabel('Sum-squared-error')
plt.title('Adaline - Learning rate 0.0005')
plt.show()

plot_decision_regions(X, y, clf=ada)
plt.title('Adaline - Gradient Descent')
plt.show()
```


Adaline - Learning rate 0.00055
Sum-squared-error
Iterations
75
70
65
60
55
50
45
40
2
4
6
8
10


Adaline - Learning rate 0.0005
Sum-squared-error
Iterations
50
45
40
35
30
25
20
2
4
6
8
10


Adaline - Gradient Descent
-1
1
6
5
4
3
2
1
0
4
5
6
7
8


Adaline - Gradient Descent
-1
1
6
5
4
3
2
1
0
4
5
6
7
8

Feature Standardization

The convergence rate depends on the eigenvalue spread, defined as:

$$\chi(\boldsymbol{X}^T\boldsymbol{X}) = \frac{\lambda_{\max}}{\lambda_{\min}} \tag{36}$$

The fastest rate is attained when $\chi(\boldsymbol{X}^T\boldsymbol{X}) = 1$, or all eigenvalues are identical. In theory, we can transform $\boldsymbol{X}$ to $\boldsymbol{Y}$ such that $\chi(\boldsymbol{Y}^T\boldsymbol{Y}) = 1$, but it is a difficult task.

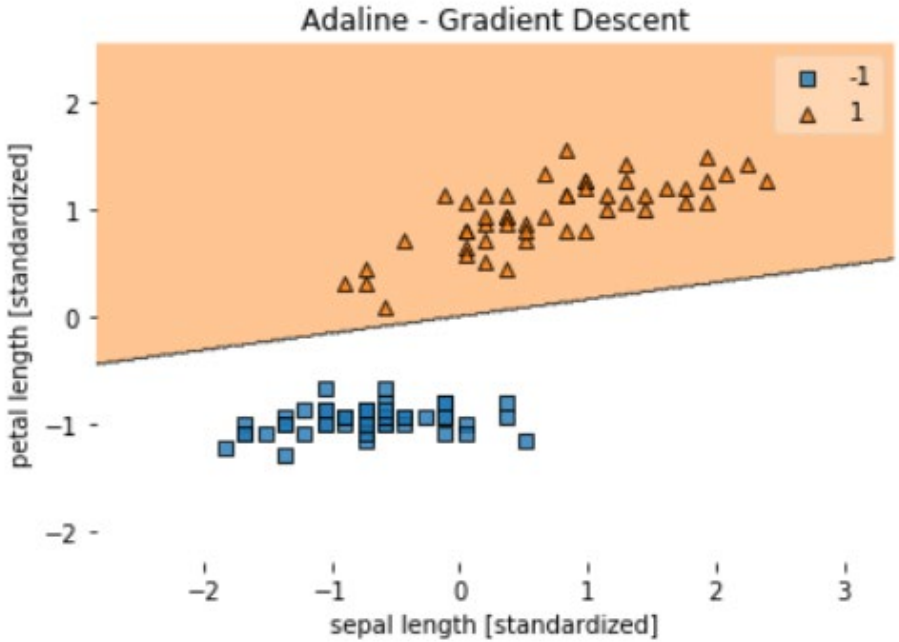
One simple way which can often increase the convergence rate is to shift and scale each feature:

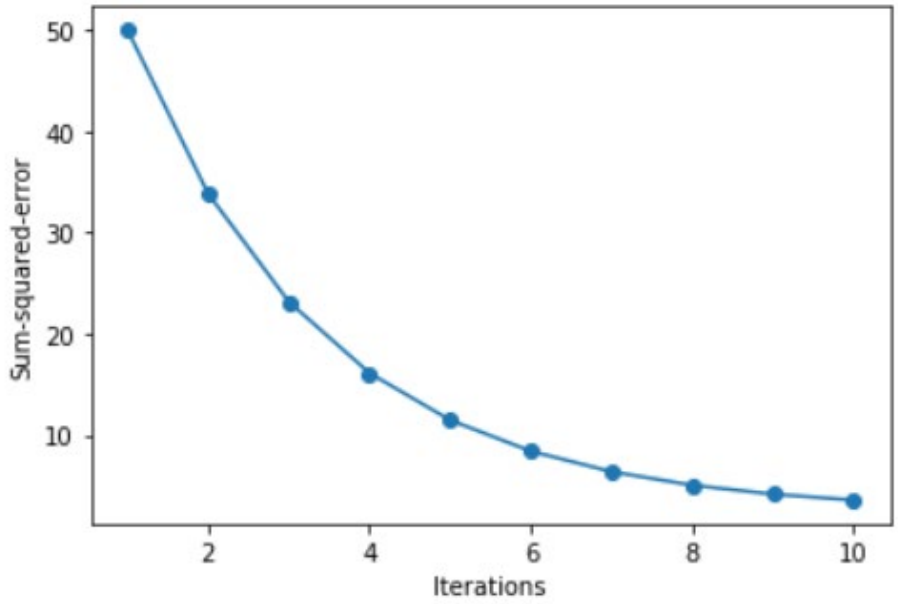
$$\bar{x}_{j,i} = \frac{x_{j,i} - \mu_j}{\sigma_j}, \ j = 1, \cdots, m, \ i = 1, \cdots, N \tag{37}$$

where μ_j and σ_j are mean and standard deviation of the jth feature. In doing so, all modified features have zero mean and unit variance.

```
# standardize features
X_std = np.copy(X)
X_std[:,0] = (X[:,0] - X[:,0].mean()) / X[:,0].std()
X_std[:,1] = (X[:,1] - X[:,1].mean()) / X[:,1].std()
import matplotlib.pyplot as plt
from mlxtend.plotting import plot_decision_regions

ada = AdalineGD(epochs=10, eta=0.01)

ada.train(X_std, y)
plot_decision_regions(X_std, y, clf=ada)
plt.title('Adaline - Gradient Descent')
plt.xlabel('sepal length [standardized]')
plt.ylabel('petal length [standardized]')
plt.show()

plt.plot(range(1, len( ada.cost_)+1), ada.cost_, marker='o')
plt.xlabel('Iterations')
plt.ylabel('Sum-squared-error')
plt.show()
```

All data points are now closer to the origin and we see that faster convergence is attained.


Adaline - Gradient Descent
-1
1
petal length [standardized]
sepal length [standardized]
2
1
0
−1
−2
−2
−1
0
1
2
3


Sum-squared-error
Iterations
50
40
30
20
10
2
4
6
8
10

Online Mode Solution for Linear Model

In the previously discussed iterative techniques, we minimize the LS cost function in each step:

$$J_{\text{LS}}(\boldsymbol{w}) = \sum_{i=1}^{N} (d_i - \boldsymbol{w}^T \boldsymbol{x}_i)^2 \tag{38}$$

Instead we can minimize only one of the N squared terms in each step, and only one training sample pair is involved:

$$(d_i - \boldsymbol{w}^T \boldsymbol{x}_i)^2 \Rightarrow \frac{\partial (d_i - \boldsymbol{w}^T \boldsymbol{x}_i)^2}{\partial \boldsymbol{w}} = -2(d_i - \boldsymbol{w}^T \boldsymbol{x}_i)\boldsymbol{x}_i \tag{39}$$

That is to say, it just removes the sign operator in the perceptron algorithm of (7):

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + 0.5\eta[d_t - (\boldsymbol{w}_t^T \boldsymbol{x}_t)]\boldsymbol{x}_t \tag{40}$$

Using the current notation, this online algorithm is summarized as:

1. Initialize $\boldsymbol{w}^{(0)} = \mathbf{0}$
2. For $k = 1, 2, \cdots$ ($\{\boldsymbol{x}_k\} = \boldsymbol{x}_1, \cdots, \boldsymbol{x}_N, \boldsymbol{x}_1, \cdots, \boldsymbol{x}_N, \boldsymbol{x}_1, \cdots$)

   Update the weight vector as:

$$\boldsymbol{w}^{(k+1)} = \boldsymbol{w}^{(k)} + \mu[d_k - (\boldsymbol{w}^{(k)T}\boldsymbol{x}_k)]\boldsymbol{x}_k \tag{41}$$

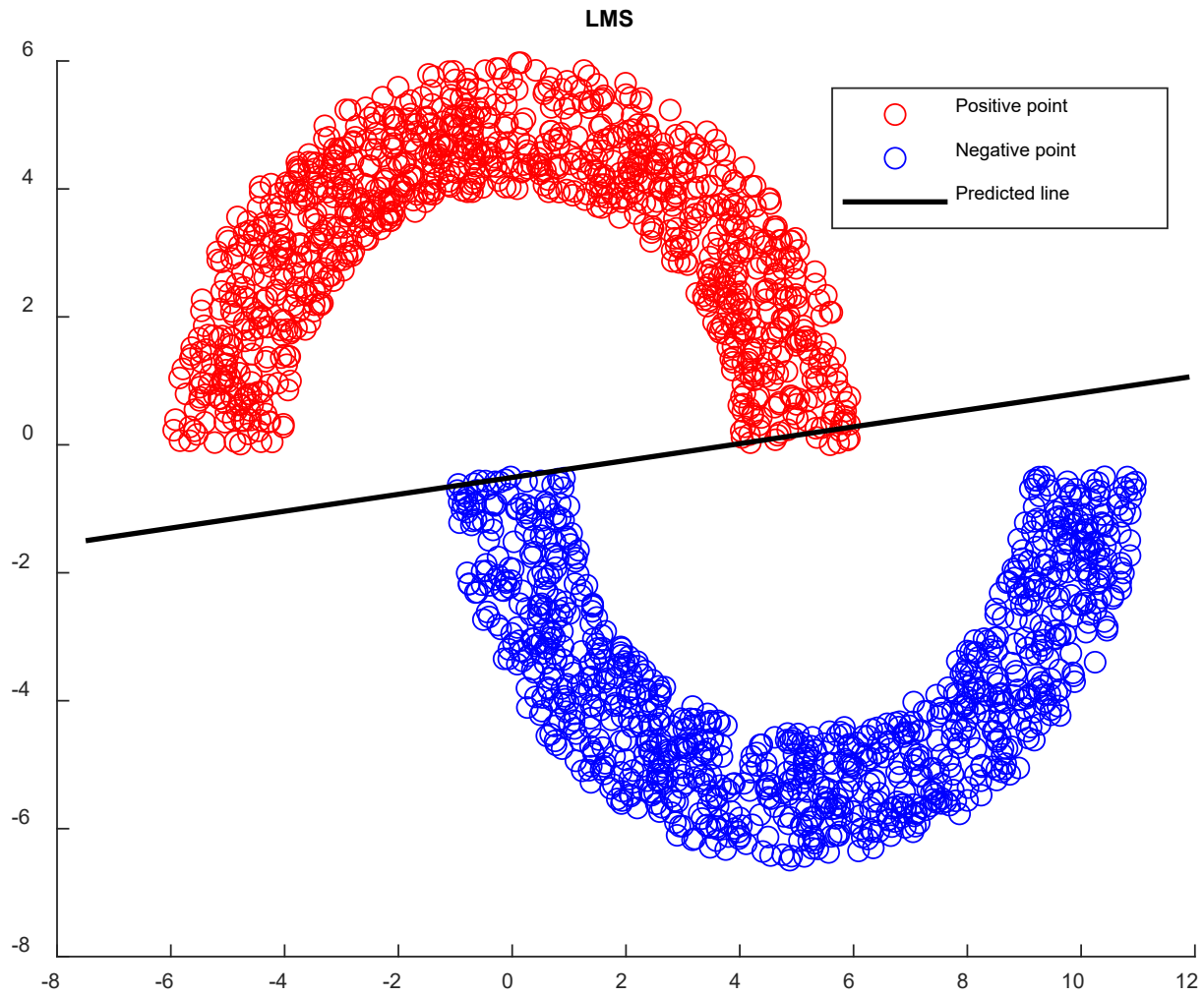
   until $\boldsymbol{w}^{(k)}$ converges.

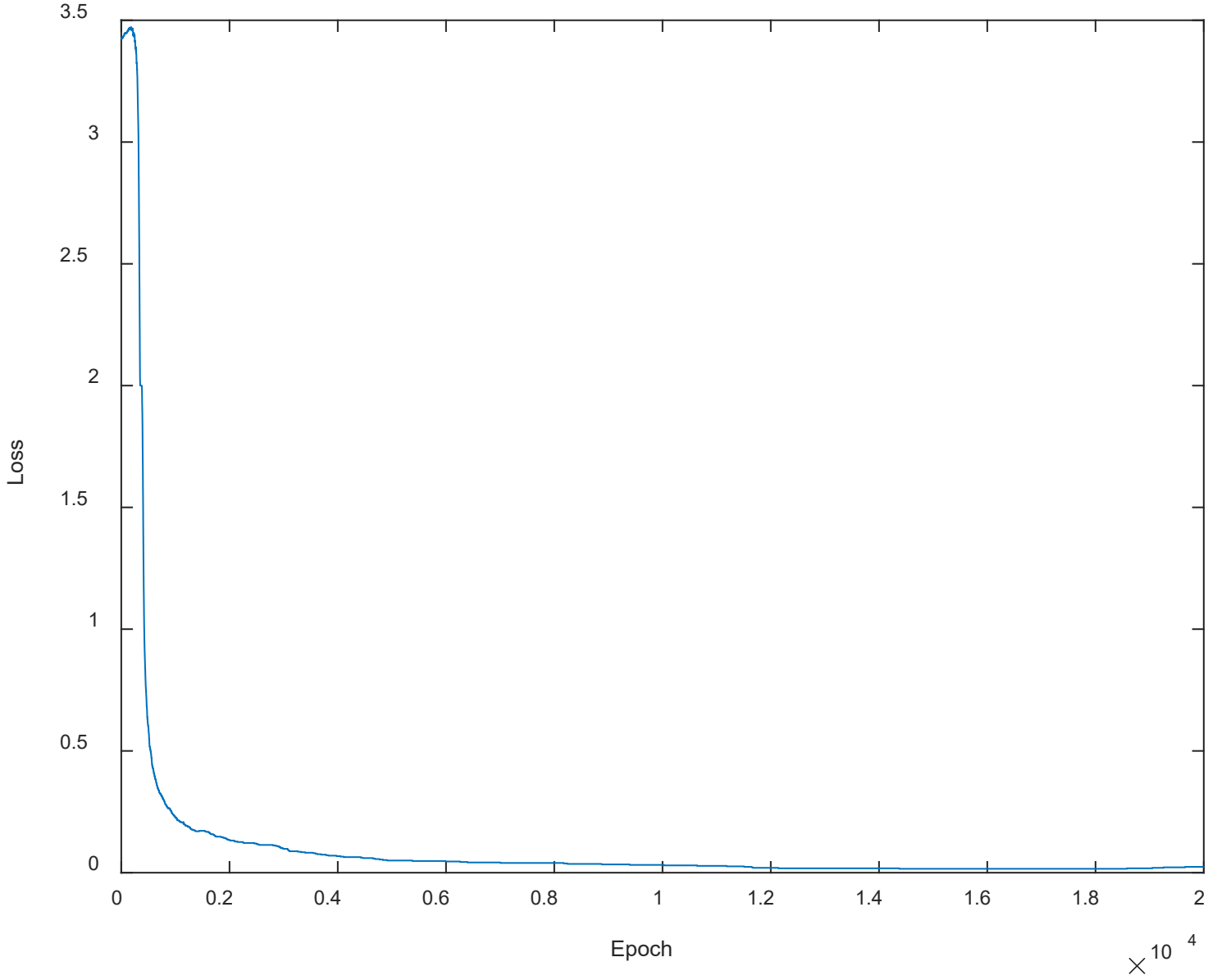
It can be proved that $\boldsymbol{w}^{(k)} \to \boldsymbol{w}^{\star}$ upon convergence, which is the LS solution.

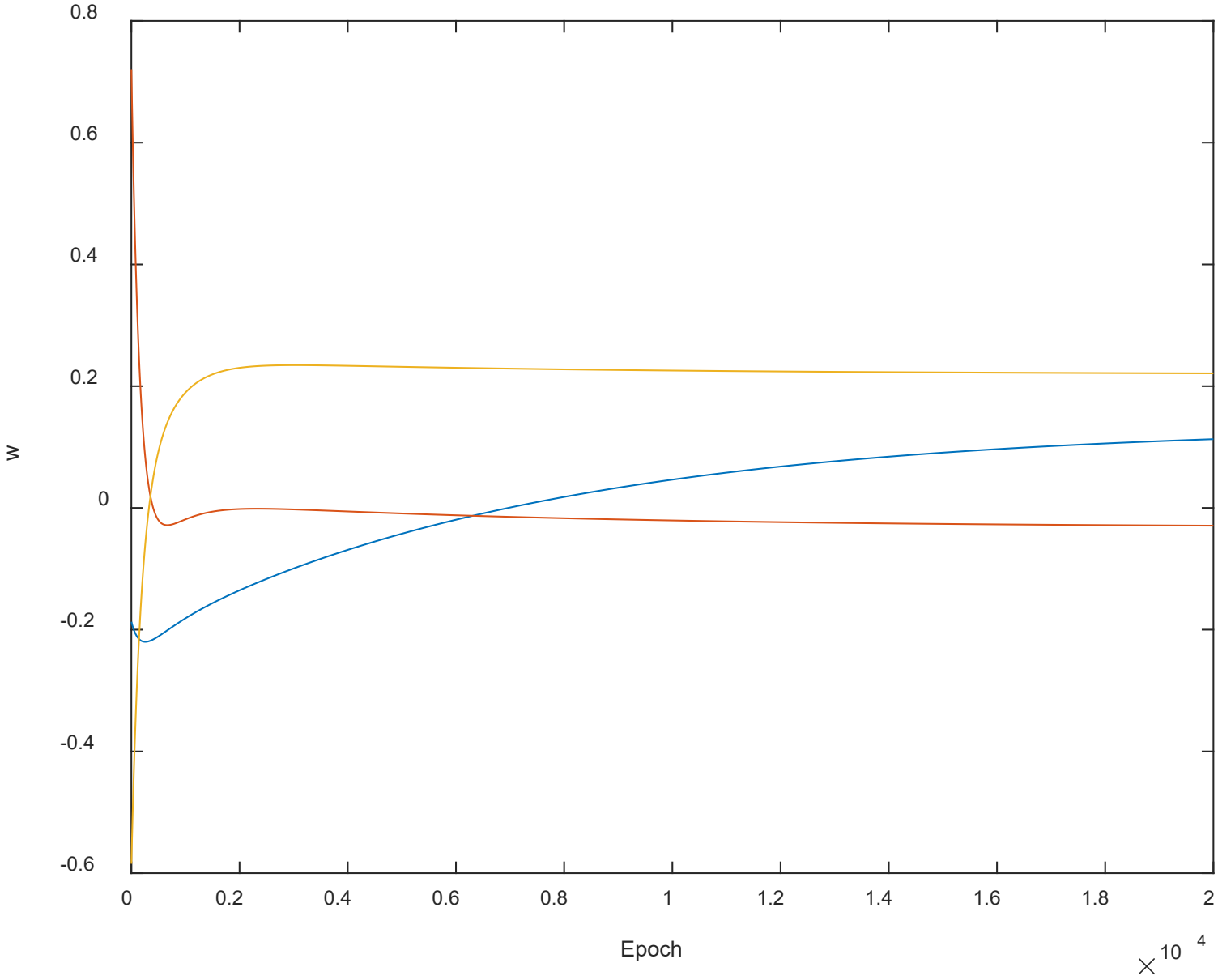
This iterative rule which updates one sample at each iteration, is also referred to as ADALINE (adaptive linear, adaptive linear element, or adaptive linear neuron), Widrow-Hoff delta rule, and least-mean-square (LMS) algorithm.

Demo.m

```
clear
close all
r = 5;
w = 2;
d = 0.5;
N = 1000;
```

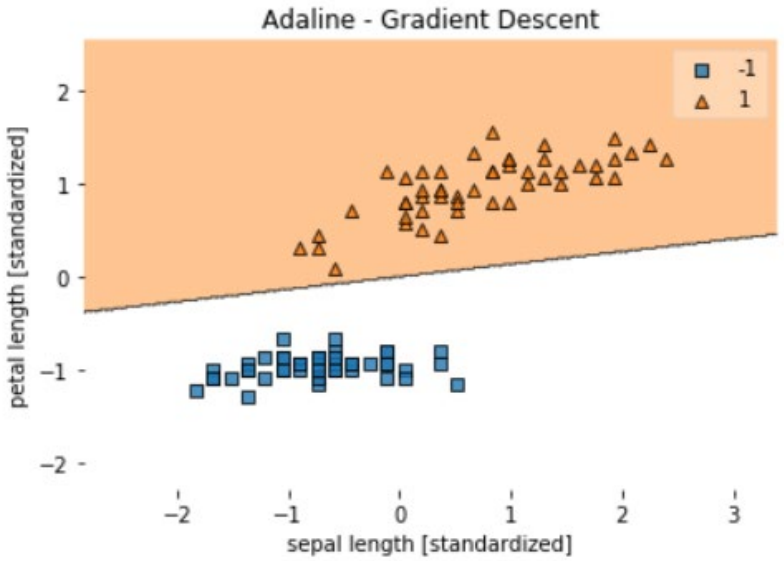


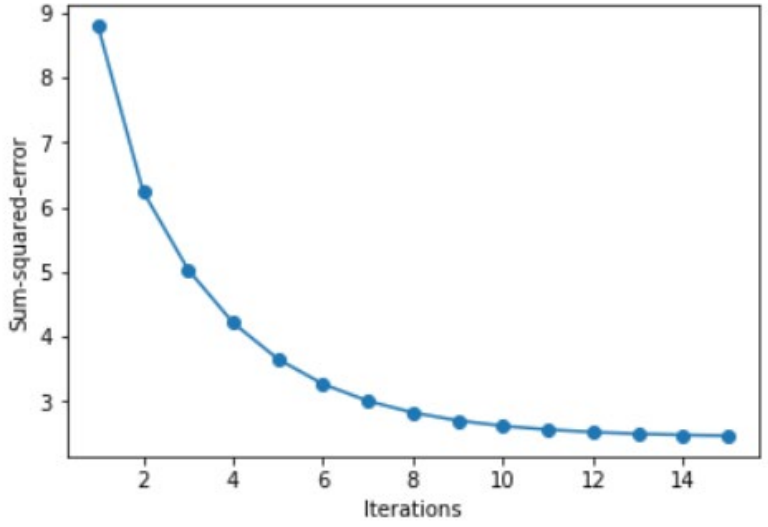

3.5
3
2.5
2
1.5
1
0.5
0
Loss
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
1.8
2
Epoch
×10^4


0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
w
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
1.8
2
Epoch
× 10 4

We also repeat the iris classification problem with feature standardization using the LMS algorithm.

```python
import numpy as np

class AdalineSGD(object):

    def __init__(self, eta=0.01, epochs=50):
        self.eta = eta
        self.epochs = epochs

    def train(self, X, y, reinitialize_weights=True):

        if reinitialize_weights:
            self.w_ = np.zeros(1 + X.shape[1])
        self.cost_ = []

        for i in range(self.epochs):
            for xi, target in zip(X, y):
                output = self.net_input(xi)
                error = (target - output)
                self.w_[1:] += self.eta * xi.dot(error)
                self.w_[0] += self.eta * error

            cost = ((y - self.activation(X))**2).sum() / 2.0
            self.cost_.append(cost)
        return self

    def net_input(self, X):
        return np.dot(X, self.w_[1:]) + self.w_[0]

    def activation(self, X):
        return self.net_input(X)

    def predict(self, X):
        return np.where(self.activation(X) >= 0.0, 1, -1)
```





```python
import pandas as pd
df = pd.read_csv('iris.data', header=None)

# setosa and versicolor
y = df.iloc[0:100, 4].values
y = np.where(y == 'Iris-setosa', -1, 1)

# sepal length and petal length
X = df.iloc[0:100, [0,2]].values


X_std = np.copy(X)
X_std[:,0] = (X[:,0] - X[:,0].mean()) / X[:,0].std()
X_std[:,1] = (X[:,1] - X[:,1].mean()) / X[:,1].std()

import matplotlib.pyplot as plt
from mlxtend.plotting import plot_decision_regions
```

```python
ada = AdalineSGD(epochs=15, eta=0.01)

# shuffle data
np.random.seed(123)
idx = np.random.permutation(len(y))
X_shuffled, y_shuffled =  X_std[idx], y[idx]

# train and adaline and plot decision regions
ada.train(X_shuffled, y_shuffled)
plot_decision_regions(X_shuffled, y_shuffled, clf=ada)
plt.title('Adaline - Gradient Descent')
plt.xlabel('sepal length [standardized]')
plt.ylabel('petal length [standardized]')
plt.show()

plt.plot(range(1, len(ada.cost_)+1), ada.cost_, marker='o')
plt.xlabel('Iterations')
plt.ylabel('Sum-squared-error')
plt.show()
```

Note that there is a simple strategy here: randomize the order of the training samples. It is because the data file lists the first class and then the second class, i.e., the first half labels correspond to $d_k = 1$ while the second half are $d_k = -1$, hindering the algorithm work properly.

Least-Mean-Square Algorithm

LMS algorithm was invented by Widrow and Hoff, and the corresponding work was published in 1960:

- Widrow has been a professor at Standard University. His research focuses on adaptive signal processing, adaptive control systems, adaptive neural networks, human memory, and human-like memory for computers.
- Hoff was Widrow's Ph.D. student. Nevertheless, he is best known as the architect of the first microprocessor – Intel's 4004 released in 1971.

To clearly present the LMS algorithm, we consider the setting of time-series training data $\{\boldsymbol{x}_n, d_n\}$, n is the time index, and at time n, no future data at $n+1, n+2, \cdots$, are available, while d_n is not restricted to be $\{1, -1\}$.

We use the probabilistic model as in (13):

$$d_n = \boldsymbol{w}^T \boldsymbol{x}_n + \epsilon_n \tag{42}$$

where

$$\boldsymbol{x}_n = [x_n \; x_{n-1} \; \cdots \; x_{n-m}]^T \in \mathbb{R}^{m+1}$$

Here, d_n, $\boldsymbol{x}_n$ and ϵ_n are random while $\boldsymbol{w}$ is the constant vector to be determined.

Considering that ϵ_n is the error term, we can construct the mean square error (MSE) cost function $J_{\text{MSE}}(\boldsymbol{w})$:

$$J_{\text{MSE}}(\boldsymbol{w}) = \mathbb{E}\{\epsilon_n^2\} = \mathbb{E}\{(d_n - \boldsymbol{w}^T \boldsymbol{x}_n)^2\} \tag{43}$$

Expanding (43) yields:

$$J_{\text{MSE}}(\boldsymbol{w}) = \mathbb{E}\{d_n^2\} - 2\boldsymbol{w}^T \mathbb{E}\{d_n \boldsymbol{x}_n\} + \boldsymbol{w}^T \mathbb{E}\{\boldsymbol{x}_n \boldsymbol{x}_n^T\} \boldsymbol{w}$$

Assuming stationarity such that the statistics of d_n, $\boldsymbol{x}_n$ and ϵ_n remain unchanged for all n, and denoting $\boldsymbol{r}_{dx} = \mathbb{E}\{d_n \boldsymbol{x}_n\}$ and $\boldsymbol{R}_x = \mathbb{E}\{\boldsymbol{x}_n \boldsymbol{x}_n^T\}$, (43) becomes:

$$J_{\text{MSE}}(\boldsymbol{w}) = \mathbb{E}\{d^2\} - 2\boldsymbol{w}^T \boldsymbol{r}_{dx} + \boldsymbol{w}^T \boldsymbol{R}_x \boldsymbol{w} \tag{44}$$

Differentiating $J_{\text{MSE}}(\boldsymbol{w})$ w.r.t. $\boldsymbol{w}$, and setting the result to 0, we obtain the minimum MSE (MMSE) or Wiener solution $\hat{\boldsymbol{w}}_{\text{MMSE}}$:

$$-2\boldsymbol{r}_{dx} + 2\boldsymbol{R}_x \hat{\boldsymbol{w}}_{\text{MMSE}} \Rightarrow \hat{\boldsymbol{w}}_{\text{MMSE}} = \boldsymbol{R}_x^{-1} \boldsymbol{r}_{dx} \tag{45}$$

which is analogous to the LS solution in (25). Note that $\hat{\boldsymbol{w}}_{\text{LS}}$ can be equal to $\hat{\boldsymbol{w}}_{\text{MMSE}}$ when $N \to \infty$.

Although the Wiener solution is optimum, what is the problem?

To practically realize the minimization of $J_{\text{MSE}}(\boldsymbol{w})$ in a computationally simple manner, Widrow and Hoff proposed minimizing the instantaneous squared error ϵ_n^2 where

$$\epsilon_n = d_n - \boldsymbol{x}_n^T \boldsymbol{w}^{(n)}$$

instead of $\mathbb{E}\{\epsilon_n^2\}$ via steepest descent:

$$\begin{aligned}
\boldsymbol{w}^{(n+1)} &= \boldsymbol{w}^{(n)} - 0.5\mu \frac{\partial \epsilon_n^2}{\partial \boldsymbol{w}^{(n)}} \\
&= \boldsymbol{w}^{(n)} - 0.5\mu \frac{\partial (d_n - (\boldsymbol{w}^{(n)})^T \boldsymbol{x}_n)^2}{\partial \boldsymbol{w}^{(n)}} \\
&= \boldsymbol{w}^{(n)} + \mu (d_n - (\boldsymbol{w}^{(n)})^T \boldsymbol{x}_n) \boldsymbol{x}_n \\
&= \boldsymbol{w}^{(n)} + \mu \epsilon_n \boldsymbol{x}_n
\end{aligned} \tag{46}$$

Comparing with (39), we can explain why the algorithm in (40) or (41) is called the LMS algorithm.

We use a system identification example to illustrate the applicability of (46).

Consider a linear time-invariant system with input $x(n)$ and impulse response $h(n)$, the system output $d(n)$ is:

$$d(n) = x(n) \otimes h(n) = \sum_{m=-\infty}^{\infty} x(m)h(n-m)$$

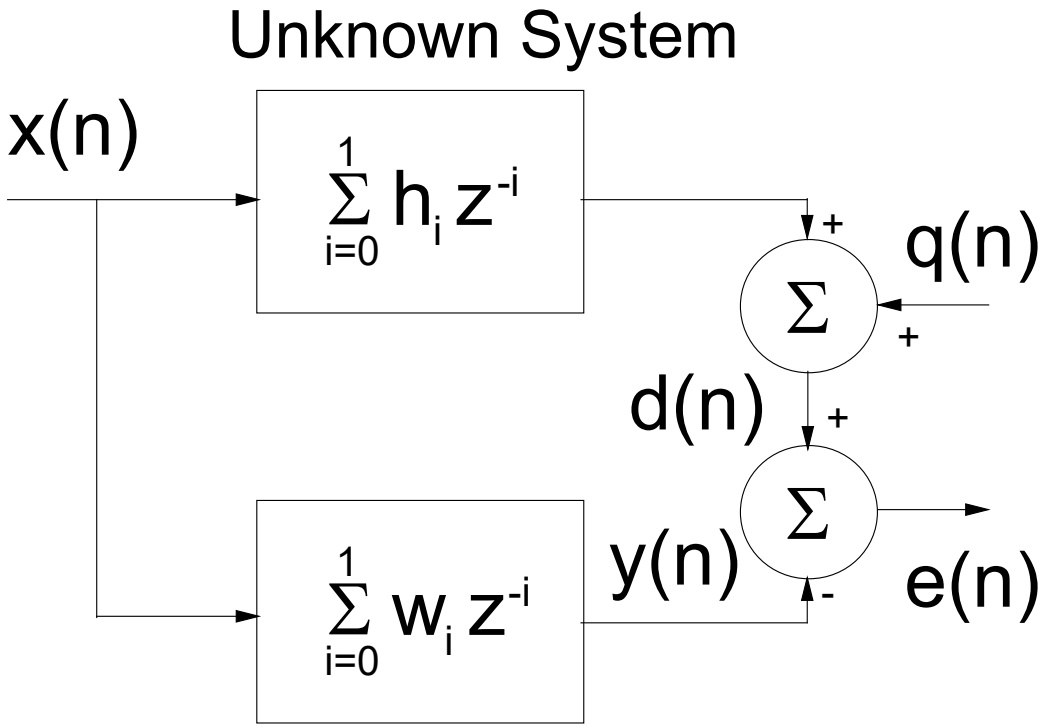
where $\otimes$ denotes the convolution operator. For a simple finite impulse response system, say, $\boldsymbol{h} = [h(0)\ h(1)]^T$, then:

$$d(n) = h(0)x(n) + h(1)x(n-1)$$

In the presence of noise $q(n)$, the output is modelled as:

$$d(n) = h(0)x(n) + h(1)x(n-1) + q(n)$$

Given the time sequences of $x(n)$ and $d(n)$, we want to find $[h(0)\ h(1)]^T$ using $[w(0)\ w(1)]^T$.



$$y(n) = w_0(n)x(n) + w_1(n)x(n-1)$$
$$e(n) = d(n) - y(n) = d(n) - w_0(n)x(n) - w_1(n)x(n-1)$$

According to (46), the LMS algorithm is:

$$w_0(n+1) = w_0(n) + \mu e(n)x(n)$$
$$w_1(n+1) = w_1(n) + \mu e(n)x(n-1)$$

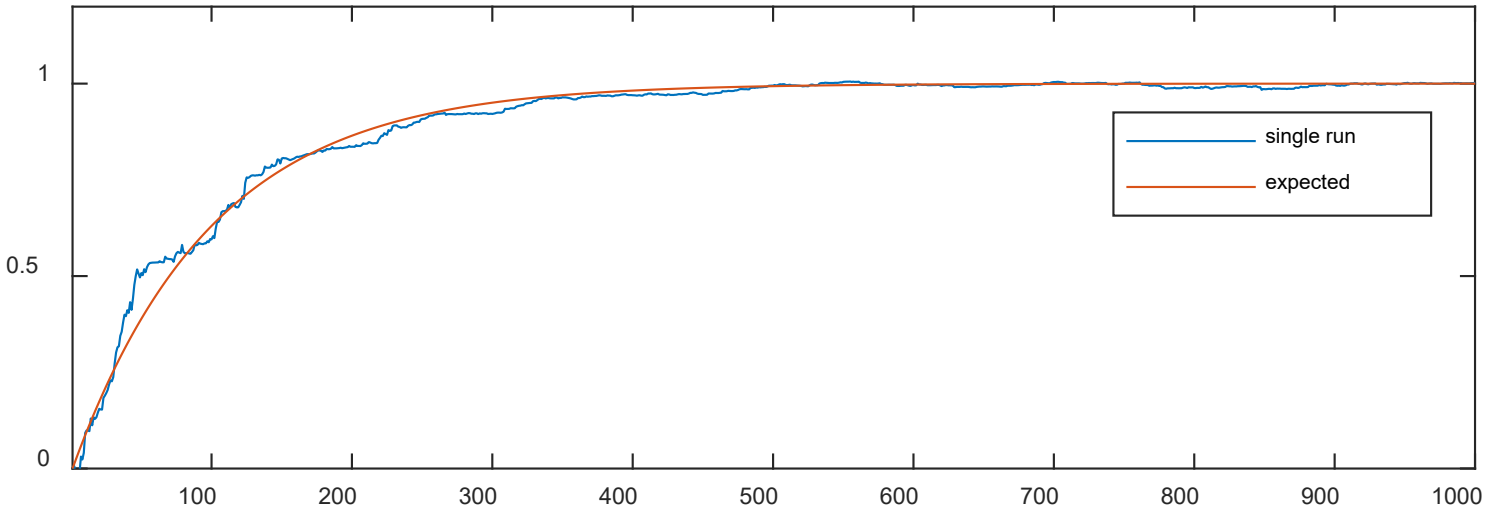
Here, we use zero-mean Gaussian IID $x(n) \sim \mathcal{N}(0, \sigma_x^2)$ and $q(n) \sim \mathcal{N}(0, \sigma_q^2)$ for data generation. Then we have:

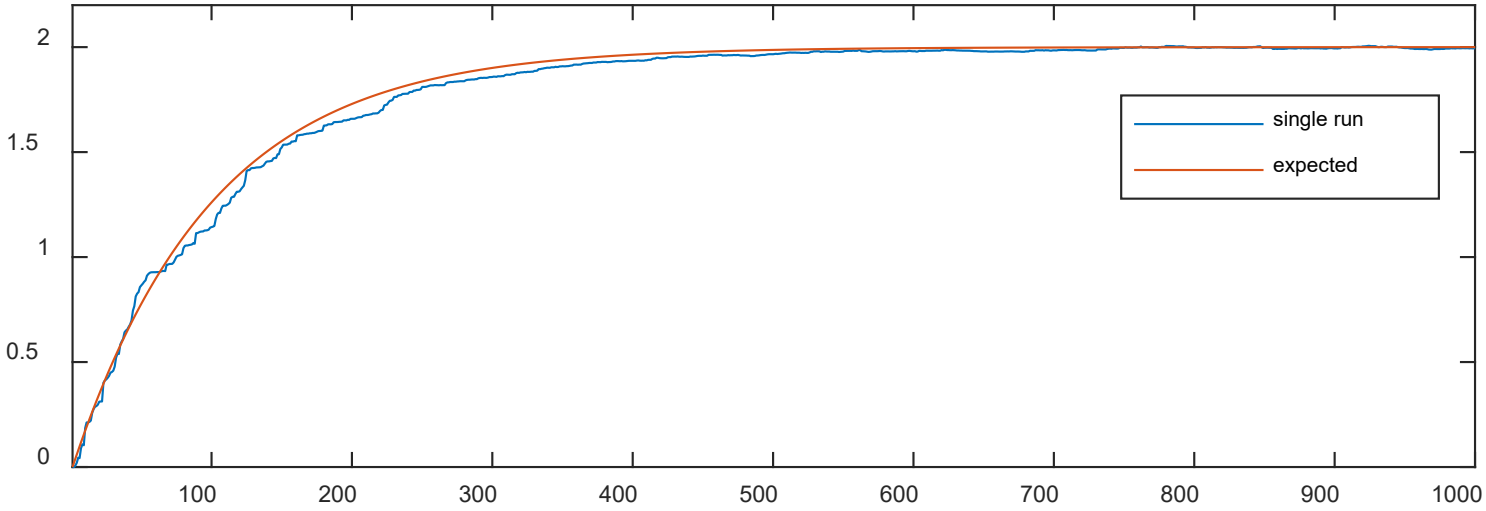
$$\mathbb{E}\{x^2(n)\} = \mathbb{E}\{x^2(n-1)\} = \sigma_x^2$$
$$\mathbb{E}\{x(n)x(n-1)\} = \mathbb{E}\{x(n)\}\mathbb{E}\{x(n-1)\} = 0$$
$$\mathbb{E}\{x(n)q(n)\} = \mathbb{E}\{x(n)\}\mathbb{E}\{q(n)\} = 0$$

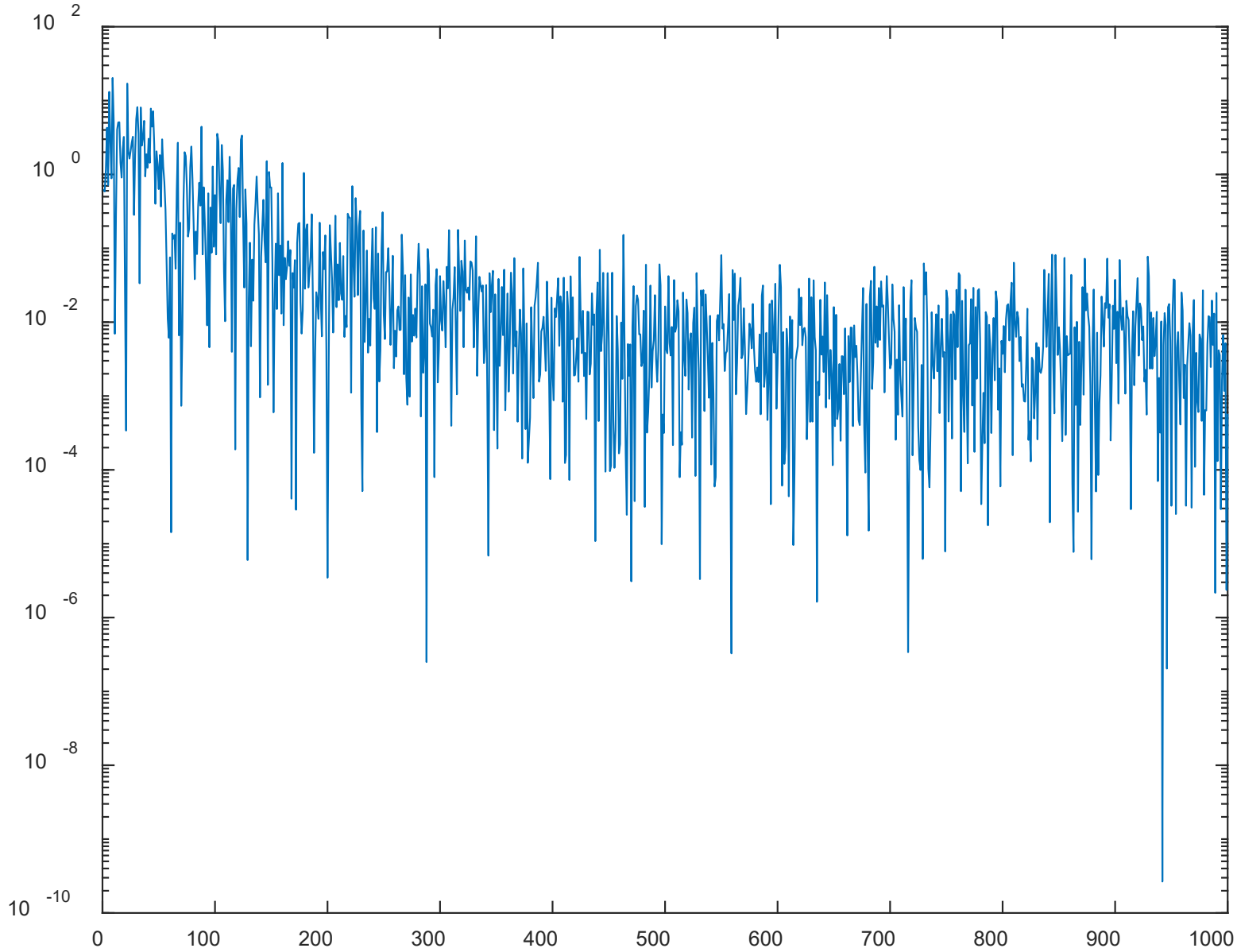
no.m | LMS_sys_ident.m

```
clear all
N=1000; % number of iterations
np = 0.01; %noise power
sp = 1; %signal power
h=[1 2]; %impulse response
x = sqrt(sp).*randn(1,N); %input signal
d = conv(x,h);
d = d(1:N) + sqrt(np).*randn(1,N); %output signal with noise
w0(1) = 0;
w1(1) = 0;
mu = 0.01;
```

```
for n=2:N
    y(n) = w0(n)*x(n) + w1(n)*x(n-1);
    e(n) = d(n) - y(n);
    w0(n+1) = w0(n) + mu*e(n)*x(n);
    w1(n+1) = w1(n) + mu*e(n)*x(n-1);
end

n = 1:N+1;
Ew0=h(1)+(1-sp*mu).^(n-1).*(w0(1)-h(1)); %expected value
Ew1=h(2)+(1-sp*mu).^(n-1).*(w1(1)-h(2)); %expected value
subplot(2,1,1)
plot(n,w0, n,Ew0)
legend('single run', 'expected')
axis([1 1000 0 1.2])
subplot(2,1,2)
plot(n,w1, n, Ew1)
legend('single run', 'expected')
axis([1 1000 0 2.2])
figure(2)
subplot(1,1,1)
n = 1:N;
semilogy(n,e.*e);
```


1
0.5
0
100
200
300
400
500
600
700
800
900
1000
single run
expected


2
1.5
1
0.5
0
100
200
300
400
500
600
700
800
900
1000
single run
expected


10 2
10 0
10 -2
10 -4
10 -6
10 -8
10 -10
0
100
200
300
400
500
600
700
800
900
1000

Taking the expected value of the first updating rule:

$$\mathbb{E}\{w_0(n+1)\} = \mathbb{E}\{w_0(n)\} + \mu\mathbb{E}\{e(n)x(n)\}$$

Investigating $\mathbb{E}\{e(n)x(n)\}$:

$$\begin{aligned}
\mathbb{E}\{e(n)x(n)\} &= \mathbb{E}\{[d(n) - w_0(n)x(n) - w_1(n)x(n-1)]x(n)\} \\
&= \mathbb{E}\{(h(0) - w_0(n))x(n) + (h(1) - w_1(n))x(n-1) + q(n)]x(n)\} \\
&= \mathbb{E}\{(h(0) - w_0(n))x(n)x(n)\} \\
&= (h(0) - \mathbb{E}\{w_0(n)\})\mathbb{E}\{x^2(n)\} \\
&= \sigma_x^2(h(0) - \mathbb{E}\{w_0(n)\})
\end{aligned}$$

Note the independence of $x(n)$ and $q(n)$. Here, we also assume that $x(n)$ and $w_0(n)$ are independent.

Hence we have:

$$\begin{aligned}
\mathbb{E}\{w_0(n+1)\} &= \mathbb{E}\{w_0(n)\} + \mu\sigma_x^2(h(0) - \mathbb{E}\{w_0(n)\}) \\
&= (1 - \mu\sigma_x^2)\mathbb{E}\{w_0(n)\} + \mu\sigma_x^2 h(0) \qquad (47)
\end{aligned}$$

Following the derivation in (32), (47) can be solved as:

$$\mathbb{E}\{w_0(n+1)\} = h(0) + (1-\mu\sigma_x^2)^{n+1}(w_0(0) - h(0))$$

Similarly, we also have:

$$\mathbb{E}\{w_1(n+1)\} = h(0) + (1-\mu\sigma_x^2)^{n+1}(w_1(0) - h(1))$$

The algorithm converges in the mean sense if

$$|1-\mu\sigma_x^2| < 1 \Rightarrow 0 < \mu < \frac{2}{\sigma_x^2}$$

What is the eigenvalue spread in this problem?

References:

1. S. Haykin, *Neural Networks and Learning Machines*, Prentice Hall, 2009
2. https://www.cs.sjsu.edu/~stamp/RUA/ann.pdf
3. https://www.cs.cornell.edu/courses/cs4780/2018fa/lectures/lecturenote03.html
4. B. Widrow and S. D. Stearns, *Adaptive Signal Processing*, Prentice-Hall, 1985
5. B. Widrow and M. E. Hoff, "Adaptive switching circuits," *Proceedings of IRE WESCON Convention Record*, 1960, pp. 96-104.
6. https://engineering.stanford.edu/news/ted-hoff-birth-microprocessor-and-beyond