ESS2222

Lecture 2 - Feasibility of Learning

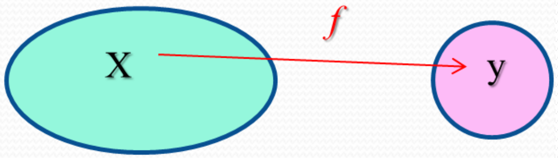
Hosein Shahnas

University of Toronto, Department of Earth Sciences,

Review of Lecture 1

Machine learning: Learning from data

Criteria: Data✓, pattern✓, no formula✓

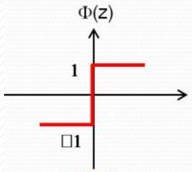


Learning model $\begin{cases} \textit{The hypothesis set} \\ H = \{h\} \quad g \in H \\ \\ \textit{The learning algorithm} \\ A \end{cases}$

f: X → y

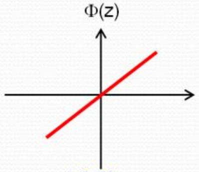
g: X → y

$h \longrightarrow g \approx f$



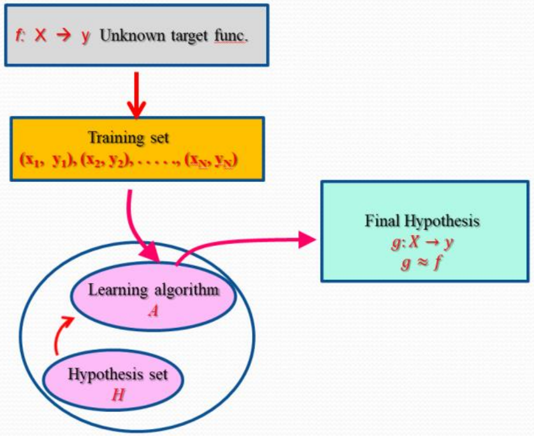
PLA

Φ(z) = sign(z)



Adeline

Φ(z) = z



Perceptron:

$$\Phi(z) = \text{sign}(z) = \begin{cases} 1, & z > 0 \\ -1, & otherwise \end{cases} \qquad \hat{y} = \Phi(z)$$

Adeline:

$$\Phi(z) = z \qquad \hat{y} = \begin{cases} 1, & \Phi(z) > 0 \\ -1, & otherwise \end{cases}$$

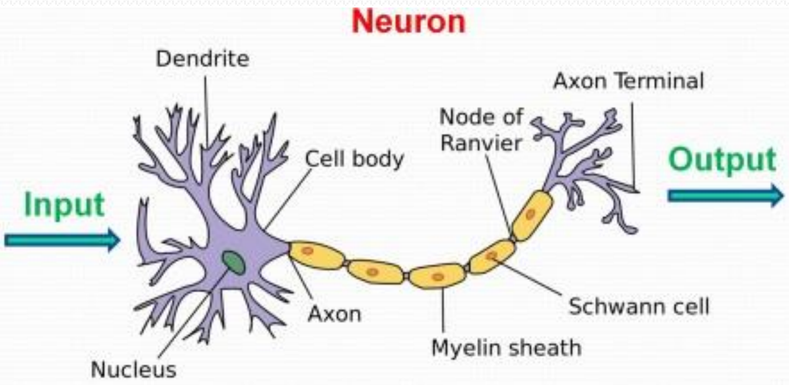
$$\boldsymbol{J(w) = \frac{1}{2} \sum_i \left(y^i - \phi(z^i)\right)^2}$$

1) Supervised learning
2) Unsupervised learning
3) Reinforcement learning

A) Classification Problem
B) Regression Problem

Feasibility of Learning - Outline

- ❑ Probabilistic Aspects of Learning
- ❑ Hoeffding's Inequality
- ❑ Generalization of Hoeffding's Inequality
- ❑ Permutation & Scaling
- ❑ Stochastic gradient decent



Is Learning Feasible?

Can we learn from a finite data set (samples) and generalize it (trough the mapping function) to the outside world?

The learned function (g)works on the sample set. How is the function outside?

The answer is the main subject of this lecture.

A Probabilistic Situation
An Experiment

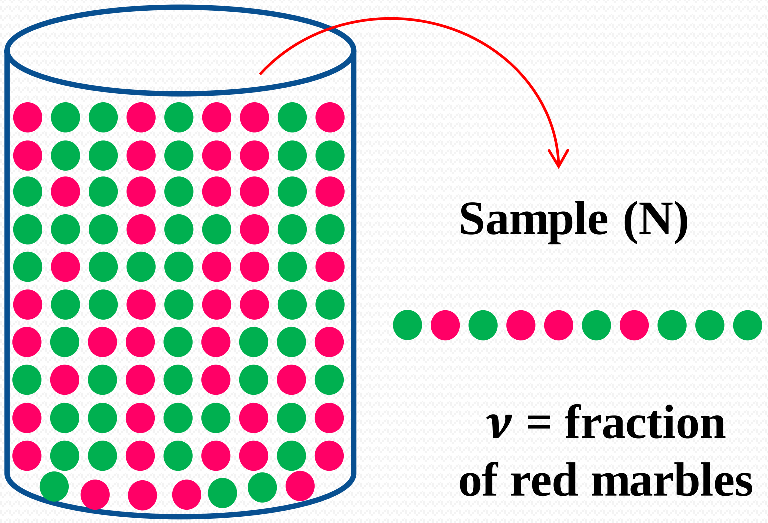
Consider a bin with green and red marbles:
Pick N marbles independently (one by one).

Assume the fraction of the red marbles in the bin is μ and the size of bin is infinite.

Then:
P(picking a red marble) = μ
P(picking a green marble) = $1 - \mu$

μ = unknown (for us) and will remain unknown.

Bin



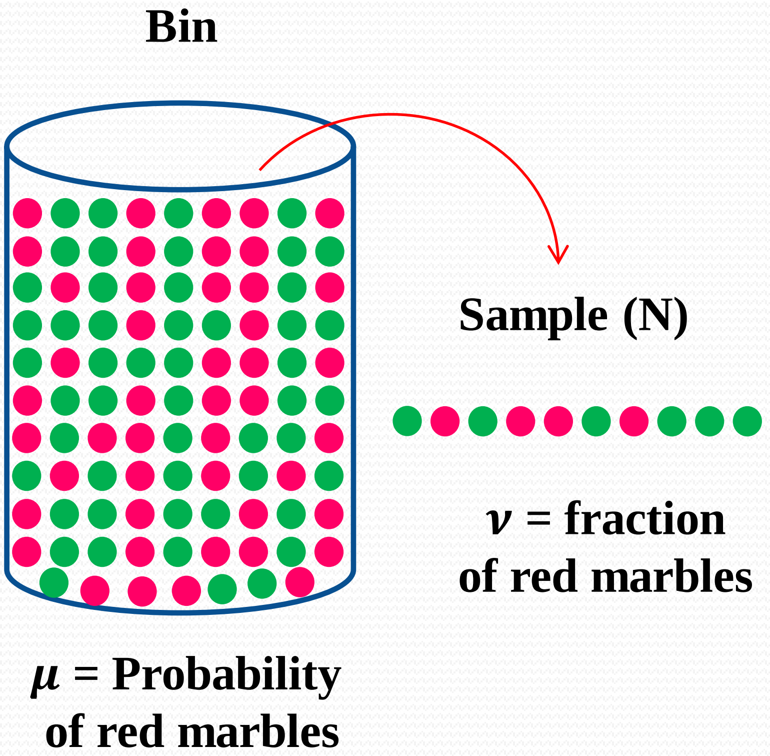
Sample (N)

ν = fraction
of red marbles

μ = Probability
of red marbles

A Probabilistic Situation
An Experiment

How ν is related to μ?
Can we say anything about μ having ν?



Bin

Sample (N)

ν = fraction of red marbles

μ = Probability of red marbles

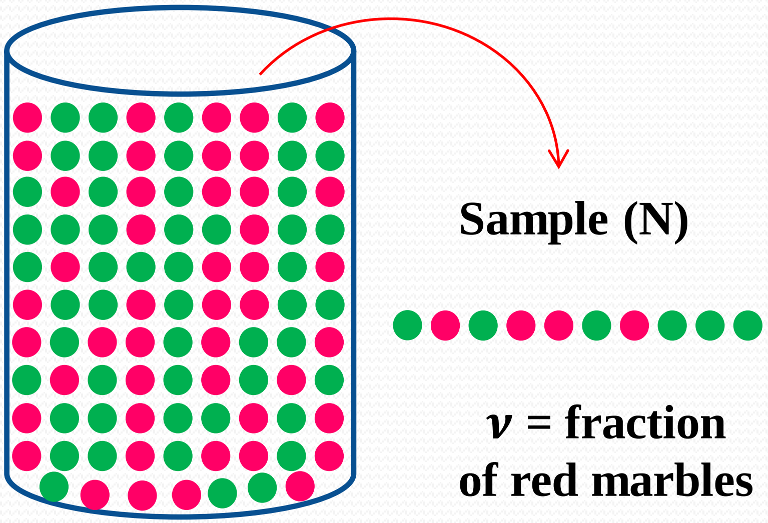
A Probabilistic Situation
An Experiment

How ν is related to μ?
Can we say anything about μ having ν?

No

Because sample can be mostly green while bin is mostly red. ●●●●●●●●●●

Bin



Sample (N)

ν = fraction of red marbles

μ = Probability of red marbles

A Probabilistic Situation
An Experiment

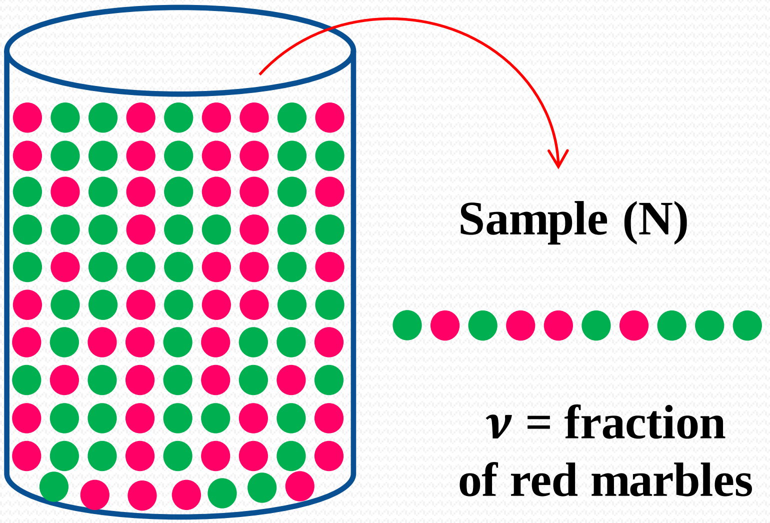
How ν is related to μ?
Can we say anything about μ having ν?

No
Because sample can be mostly green while bin is mostly red. 🟢🟢🟢🟢🟢🟢🟢🟢🟢🟢

And yes
Because if the sample is large enough, sample frequency ν is likely close to bin probability μ.

🟢🔴🟢🔴🔴🟢🔴🟢🟢🔴🟢🔴🟢🔴🔴🟢🔴🟢🟢🟢

Bin



Sample (N)

ν = fraction of red marbles

μ = Probability of red marbles

Distinction between two answers: Possible versus probable

A Probabilistic Situation
What does ν say about μ?

What does ν say about μ?

For a big sample (large N) ν is probably close to μ (within ϵ).
Mathematically:

$$\mathrm{P}[\underbrace{|\nu - \mu| > \epsilon}_{\text{bad event}}] \leq 2e^{-2\epsilon^2 N}$$

Hoeffding's inequality

Bound does not depend on μ.

A Probabilistic Situation
What does ν say about μ?

What does ν say about μ?

For a big sample (large N) ν is probably close to μ (within ϵ).
Mathematically:

$$\mathrm{P}[\underbrace{|\nu - \mu| > \epsilon}_{\text{bad event}}] \le 2e^{-2\epsilon^2 N}$$

Hoeffding's inequality

As the number of samples increases the probability of bad event decreases: P → 0

However, as ϵ → 0, P → 2

A Probabilistic Situation
What does ν say about μ?

$$\mathrm{P}[|\nu - \mu| > \epsilon] \le 2e^{-2\epsilon^2 N} \quad \forall\ \epsilon, \mathrm{N}$$ Hoeffding's inequality

- ❑ So the statement " $\mu = \nu$" is probably approximately correct (PAC).

- ❑ Despite P depends on μ, the bound ($2e^{-2\epsilon^2 N}$) does not, which is good because μ is unknown.

- ❑ Hoeffding's inequality dictates that in order to have lower tolerance (ϵ) we need large number of samples (N).

- ❑ Note that the inequality says that: $\nu \approx \mu$, because ν is affected by μ (μ: the cuase, ν the effect), however, we infer $\mu \approx \nu$ due to the symmetry in the Hoeffding's inequality.

Learning

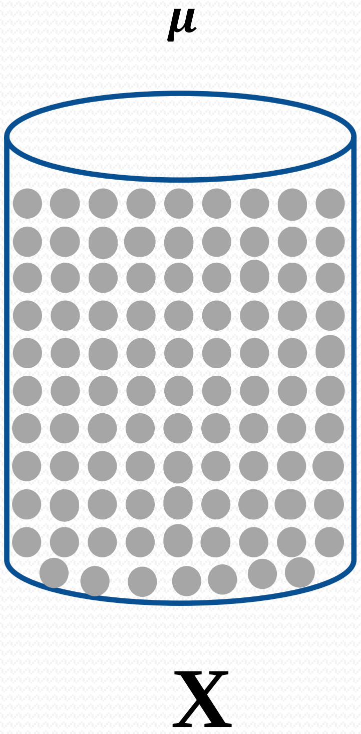
What are unknowns?

Bin: The unknown quantity is μ
Learning: The unknown is f: x →y

Each marble is a point in X space: $x \in X$

Try a single (particular) hypothesis *h (an approx. to f)*:

Hypothesis predicts correctly $h(x) = f(x)$
Hypothesis predicts wrong $h(x) \neq f(x)$



Learning

What are unknowns?

Note that *h(x)* = *f(x)* does not necessarily mean *h* = *f*

Bin: The unknown quantity is μ

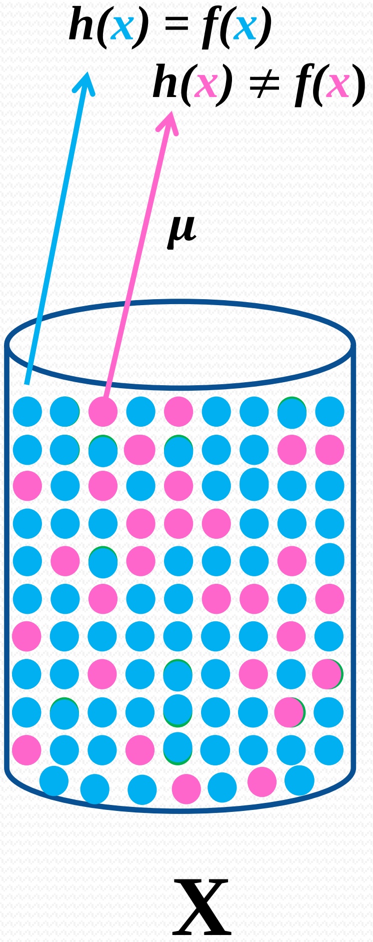
Learning: The unknown is f: x →y

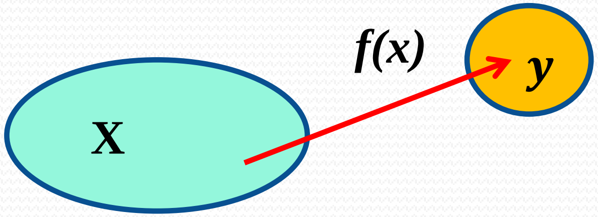
Each marble is a point in X space: x ∈ *X*

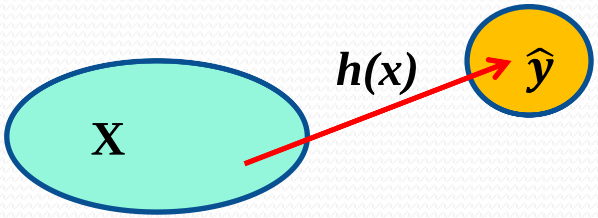
Try a single (particular) hypothesis *h (an approx. to f)*:

Hypothesis predicts correctly $h(x) = f(x)$

Hypothesis predicts wrong $h(x) \neq f(x)$



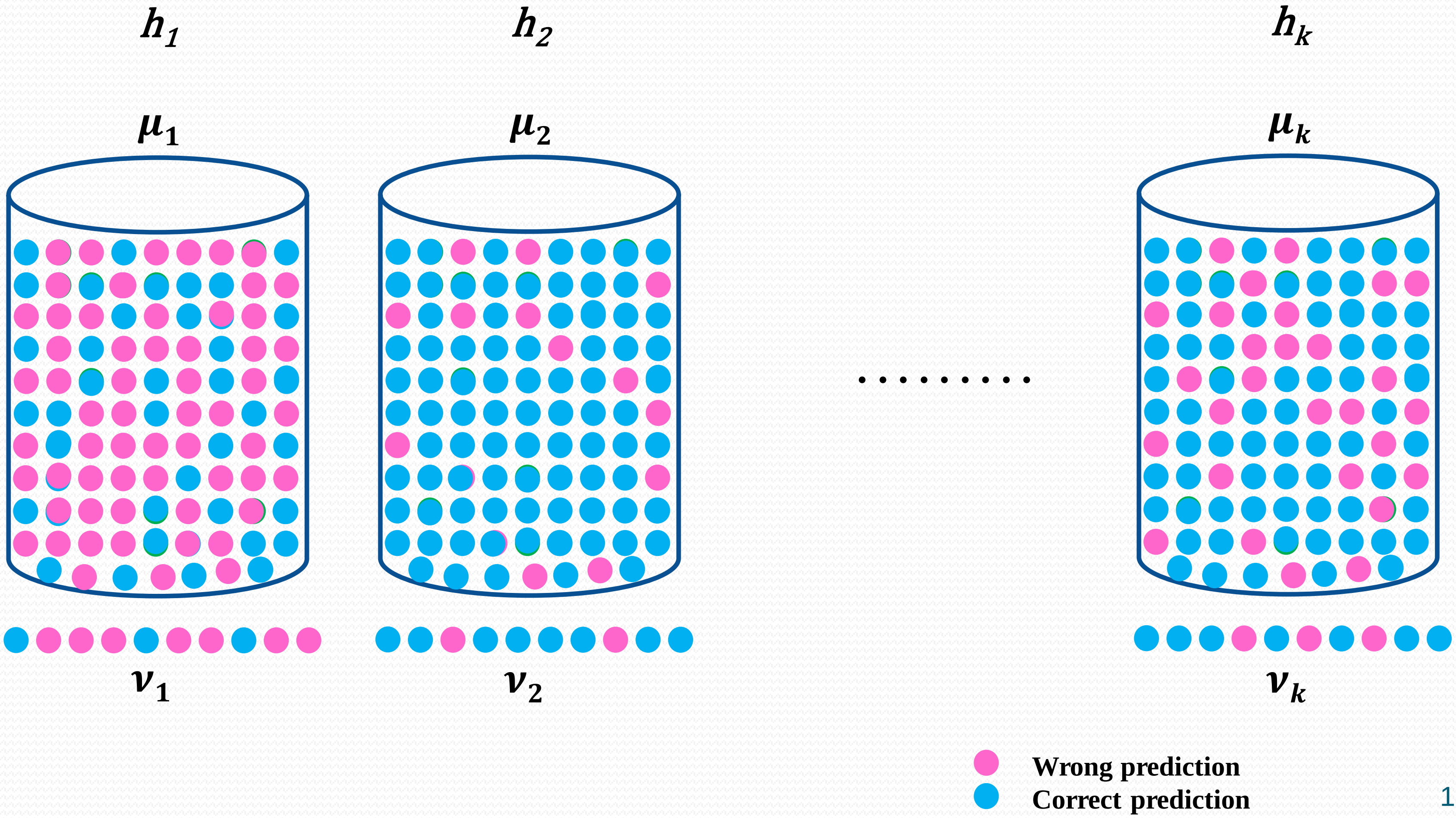




Generalization

More hypothesis:

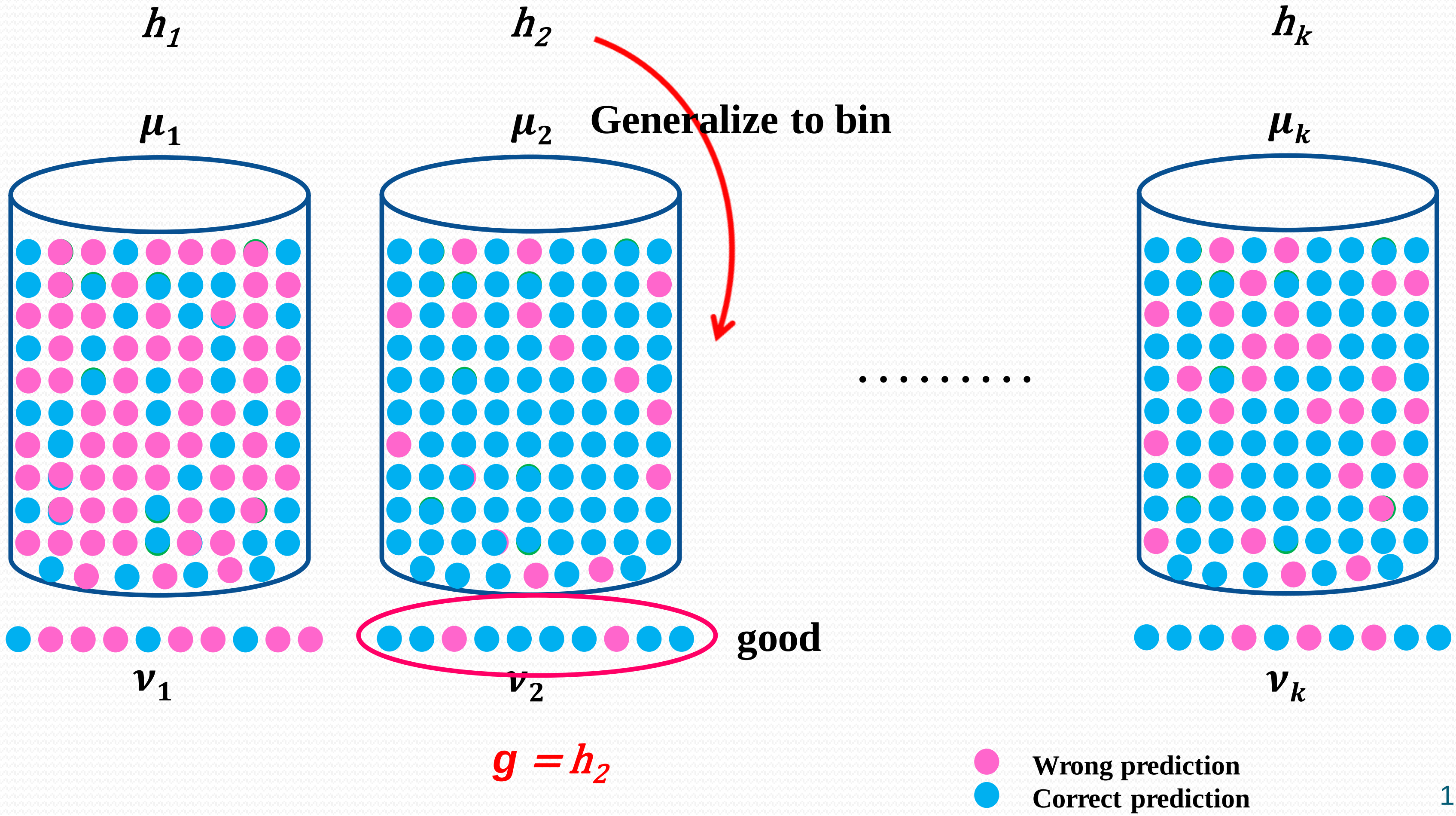
- ❑ Learning is not trying a single hypothesis *h*. Trying a single hypothesis is just a verification.
- ❑ Before generalizing ν to μ, we have to search for the set of hypothesis and find the best hypothesis.



Generalization

More hypothesis:

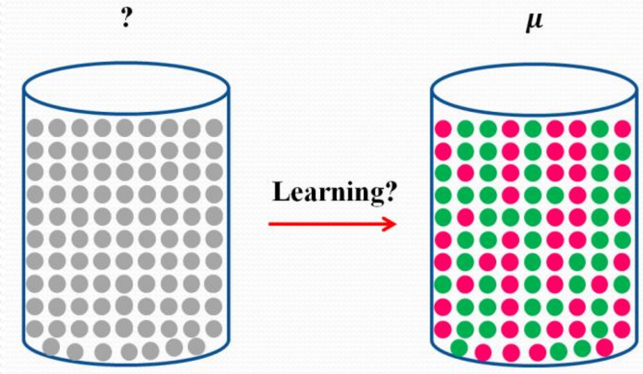
- □ Learning is not trying a single hypothesis *h*. Trying a single hypothesis is just a verification.
- □ Before generalizing ν to μ, we have to search for the set of hypothesis and find the best hypothesis.



Generalization

- Red marble
- Green marble

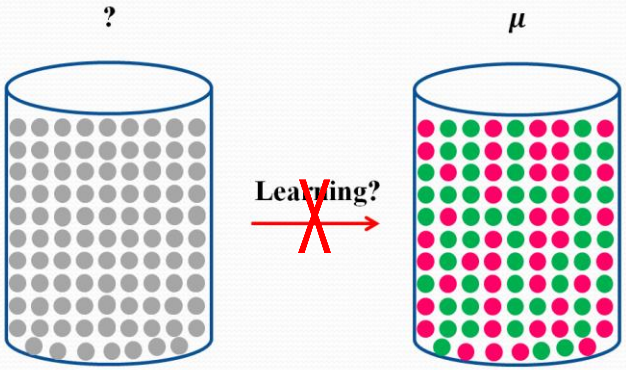
μ: unknown (exact value)



Generalization

- Red marble
- Green marble

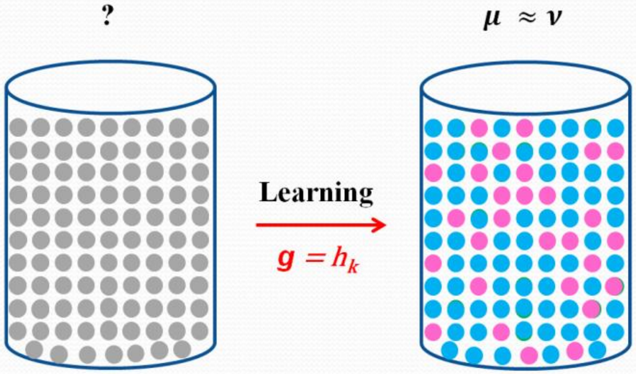
μ: unknown (exact value)



- Wrong prediction
- Correct prediction

Try to minimize the number of wrong predictions by search

$\mu \approx \nu$ with the bounded probability P



Errors

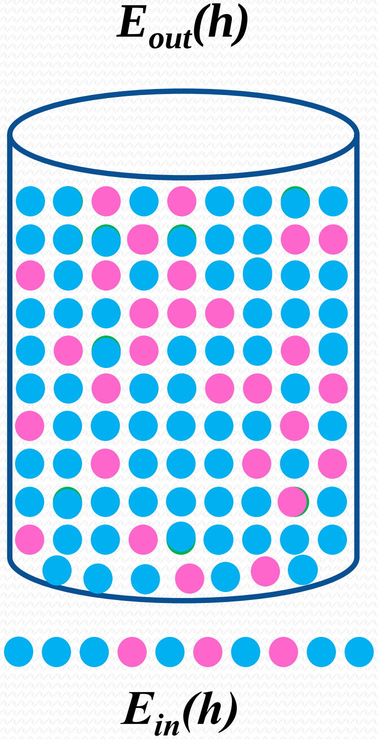
ν and μ depend on h. We introduce the error rates corresponding to ν and μ.

ν (in-sample) → $E_{in}(h)$
μ(out-sample) → $E_{out}(h)$

Then the Hoeffding's inequality becomes:

$$\mathbf{P}[|\nu - \mu| > \epsilon] \leq 2e^{-2\epsilon^2 N}$$

$$\longrightarrow \mathbf{P}[|E_{in}(h) - E_{out}(h)\ | > \epsilon] \leq 2e^{-2\epsilon^2 N}$$



Errors

ν and μ depend on h. We introduce the error rates corresponding to ν and μ.

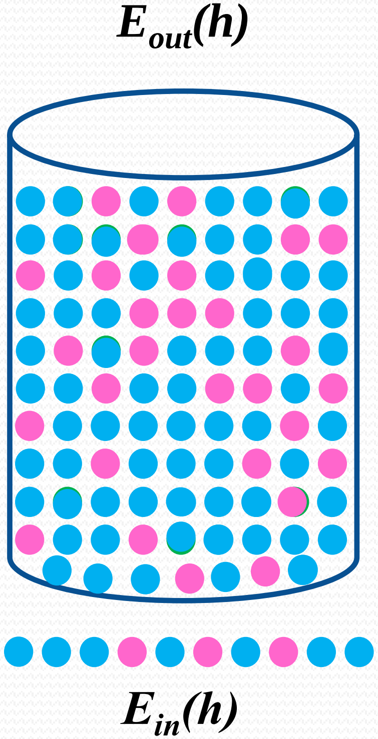
ν (in-sample) $\longrightarrow$ $E_{in}(h)$
μ(out-sample) $\longrightarrow$ $E_{out}(h)$

Then the Hoeffding's inequality becomes:

$$\mathbf{P}[|\nu - \mu| > \epsilon] \le 2e^{-2\epsilon^2 N}$$

$$\longrightarrow \mathbf{P}[|E_{in}(h) - E_{out}(h)\ | > \epsilon] \le 2e^{-2\epsilon^2 N}$$

The probability that in-sample performance deviates from out-sample performance by more than ϵ, is less than $2e^{-2\epsilon^2 N}$.

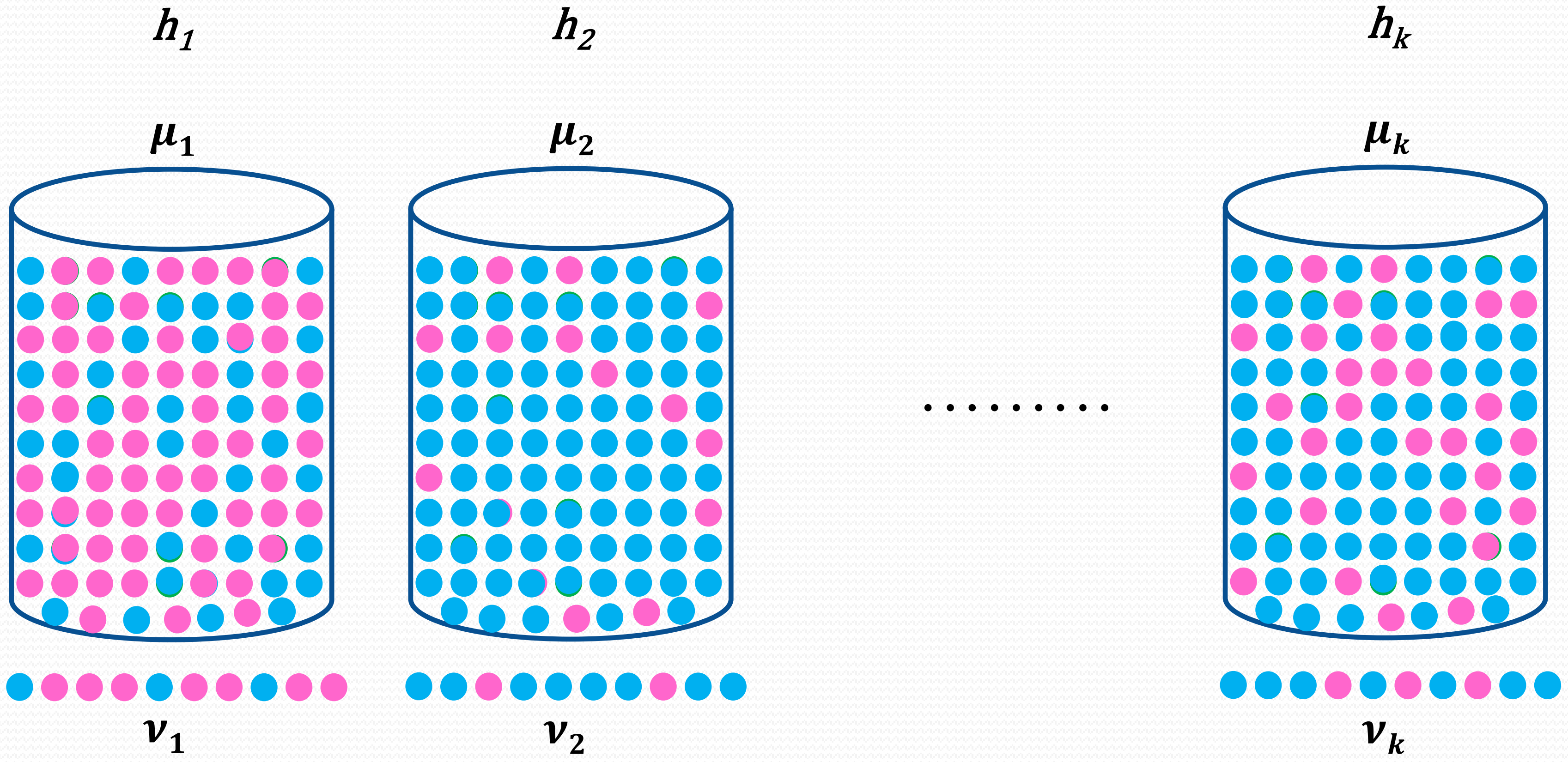


$E_{in}(h) = \frac{1}{N}\sum_1^N [\![h(x_n) \neq f(x_n)]\!]$ for classification prob.

$[\![h(x_n) \neq f(x_n)]\!] = 1$ if $h(x) \neq f(x)$, $= 0$ otherwise

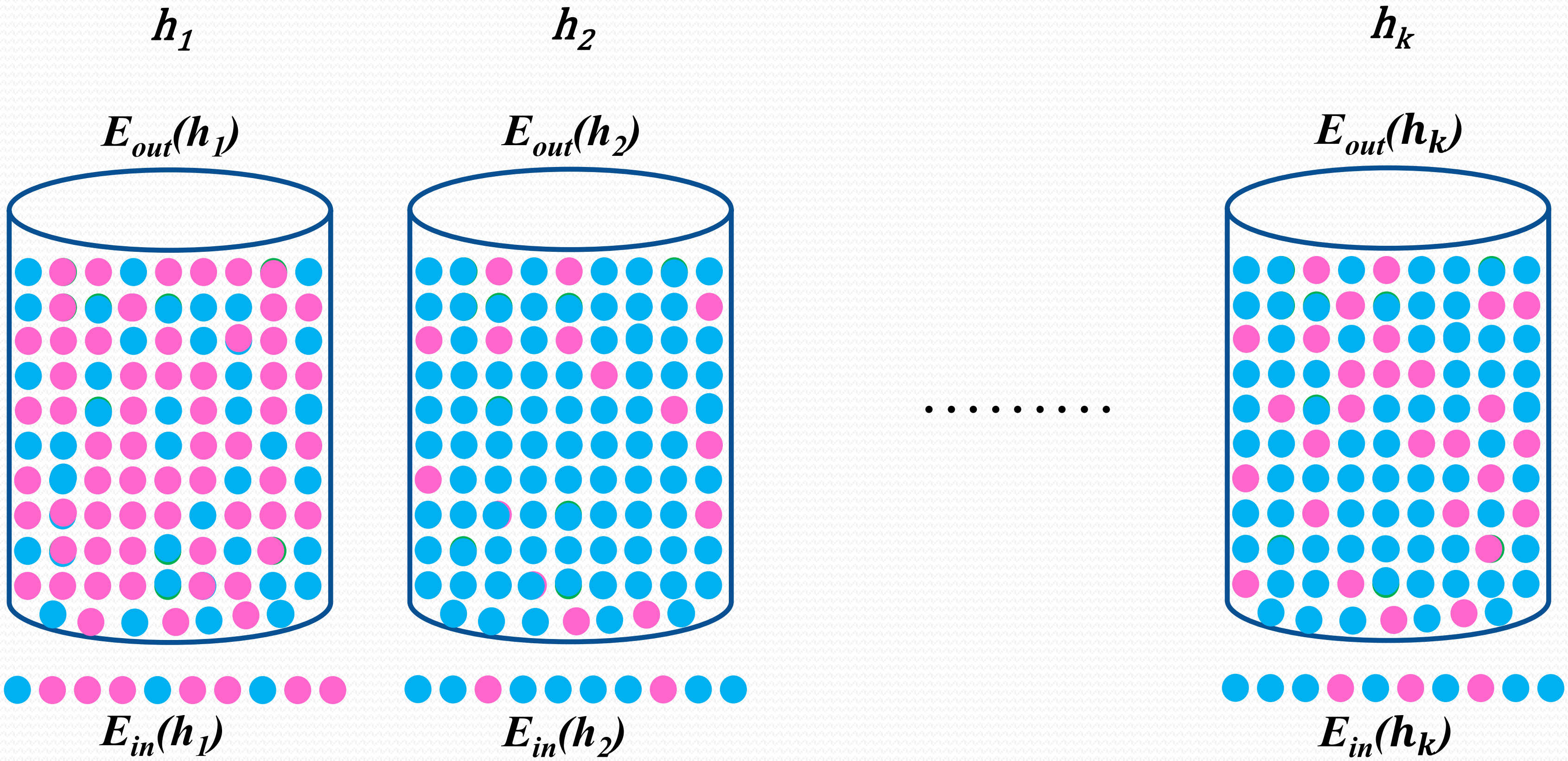
$E_{out}(h) = P\ [\![h(x) \neq f(x)]\!]$

Errors



● **Wrong prediction**
● **Correct prediction**

Errors



h_1 h_2 h_k

$E_{out}(h_1)$ $E_{out}(h_2)$ $E_{out}(h_k)$

……….

$E_{in}(h_1)$ $E_{in}(h_2)$ $E_{in}(h_k)$

● Wrong prediction

● Correct prediction

A Problem!

Hoeffding's inequality doesn’t apply to multiple bin.

Consider coin analogy

Prob. 1: If you toss a fair coin 10 times, what is the probability that you get 10 heads?

Sol.: $P = \frac{\#\ of\ events}{total\ \#\ of\ events} = \frac{1}{2^{10}}$

$P(k) = \binom{n}{k} p^k q^{n-k}$, $\qquad C^n{}_k = \binom{n}{k} = \frac{n!}{(n-k)!k!} = \frac{P^n{}_k}{k!}$,

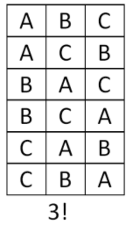
n = 10, # of flips $\qquad$ *where* $P^n{}_k = \frac{n!}{(n-k)!}$ *(k-permutations of n)*

k = 10, # of heads

p = 1/2. Prob. of success

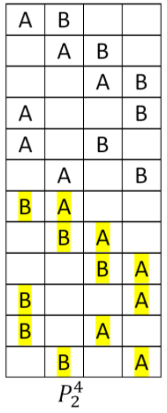
q = 1 –p = 1/2, Prob. of failure

$P(10) = \frac{1}{2^{10}} \approx 0.001$ → $P(10) \approx 0.1\%$



A	B	C
A	C	B
B	A	C
B	C	A
C	A	B
C	B	A

3!



A	B		
	A	B	
		A	B
A			B
A		B	
	A		B
B	A		
	B	A	
		B	A
B			A
B		A	
	B		A

P_2^4

A Problem!

Hoeffding's inequality doesn’t apply to multiple bin.

Consider coin analogy

Prob. 2: If you toss 1000 fair coins 10 times each, what is the probability that some coin gets 10 heads?

Sol.: Two steps:

$$P_1 = \binom{10}{0}\left(\frac{1}{2}\right)^{10}\left(1-\frac{1}{2}\right)^{0} = \frac{1}{2^{10}}, \qquad P_2 = \binom{1000}{1}(P_1)^1(1-P_1)^{999},$$

$$P_2 = \binom{1000}{1}\left(\frac{1}{2^{10}}\right)^{1}\left(1-\frac{1}{2^{10}}\right)^{999} \approx 0.368,$$

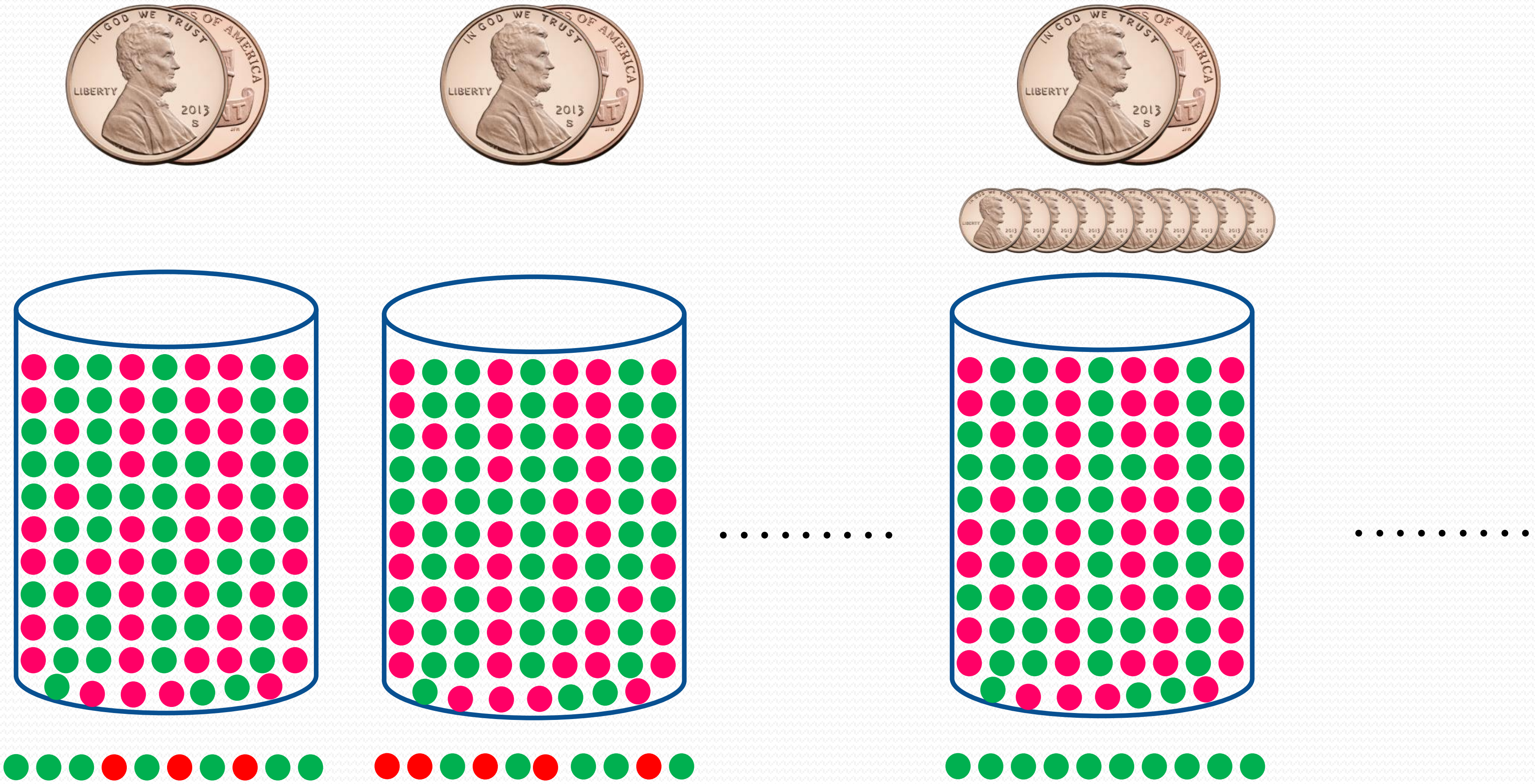
exactly 1 out of 1000 coins shows up 10 heads on 10 tosses

$$P = 1 - \underbrace{\binom{1000}{0}\left(\frac{1}{2^{10}}\right)^{0}\left(1-\frac{1}{2^{10}}\right)^{1000}}_{\text{Prob. of getting no 10-heads}} \approx 0.624,$$

at least 1 out of 1000 coins shows up 10 heads on 10 tosses

The Analogy

Now suppose μ = 0.5 (i.e. half of marbles green and half of them red).
The same probability we have in flipping a fair coin.



This is not reflecting the reality
(the real probability),

A Simple Solution

Getting 10 heads in tossing a fair coin 10 times does not reflect the reality. But if we try too hard, something bad will happen somewhere.

In mathematical language: Hoeffding's inequality applies for a single experiment (not multiple).

***P*(*g*) for the final hypothesis is less than any *P(h)* and therefore less than the union of them:**

$$\mathrm{P}[|E_{in}(g) - E_{out}(g)| > \epsilon] \leq \mathrm{P}[\ |E_{in}(h_1) - E_{out}(h_1)| > \epsilon \text{ or } |E_{in}(h_2) - E_{out}(h_2)| > \epsilon \ \ldots\ldots\ \ldots \text{ or } |E_{in}(h_k) - E_{out}(h_k)| > \epsilon\ ]$$

g*: best of *h

$$\leq \sum_{m=1}^{k} \mathrm{P}[|E_{in}(h_m) - E_{out}(h_m)| > \epsilon]$$

Union bound

This explains that why we got high probability (~0.63) for that bad event.

A Simple Solution

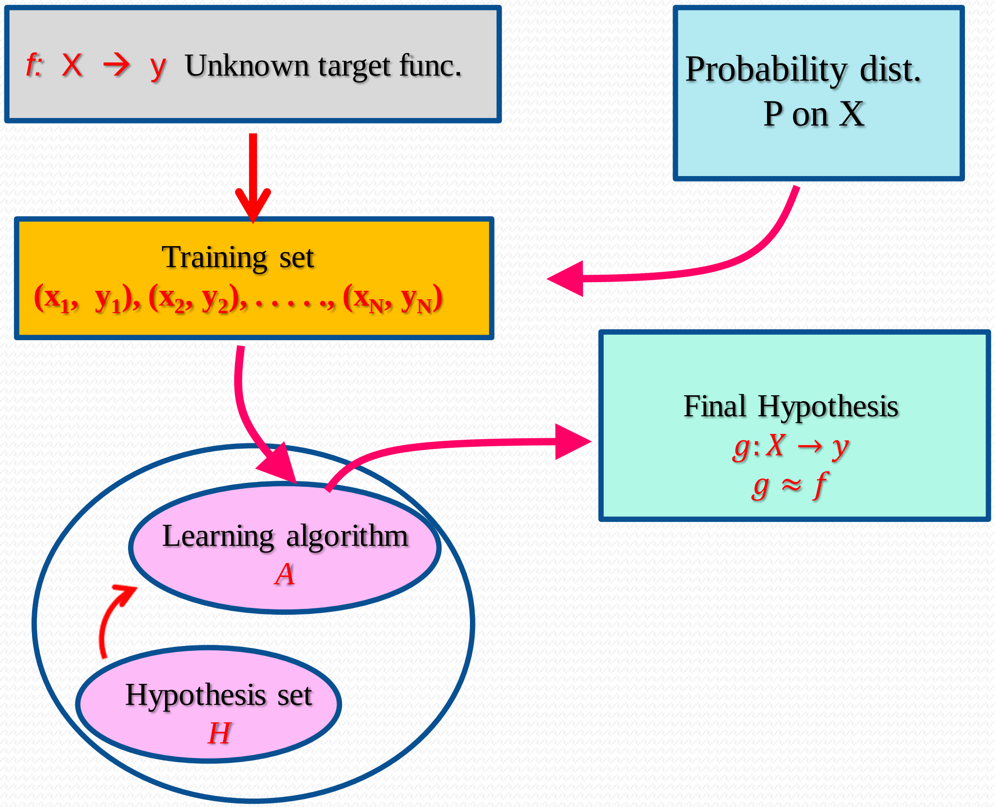
$$\mathbf{P}[|E_{in}(g) - E_{out}(g)| > \epsilon] \leq \sum_{m=1}^{k} \mathbf{P}[|E_{in}(h_m) - E_{out}(h_m)| > \epsilon]$$

$$\leq \sum_{m=1}^{k} 2e^{-2\epsilon^2 N}$$

$$\mathbf{P}[|E_{in}(g) - E_{out}(g)| > \epsilon] \leq 2Me^{-2\epsilon^2 N}$$

The factor M at the right had side of the Hoeffding's inequality increases the probability bound which is not good, however, as we will see later, the inequality can be improved.

Learning Diagram



f: X → y Unknown target func.

Probability dist.
P on X

Training set
$(x_1, y_1), (x_2, y_2), \ldots\ldots, (x_N, y_N)$

Final Hypothesis
$g: X \rightarrow y$
$g \approx f$

Learning algorithm
A

Hypothesis set
H

Learning Model

Scaling for Optimal Performance

Feature scaling:

1) Min-max normalization: $x' = \frac{x - x_{min}}{x_{max} - x_{min}}$

2) Mean normalization: $x' = \frac{x - \mu}{x_{max} - x_{min}}$

3) Normalization to a range (r_{min}, r_{max}): $x' = \frac{x - x_{min}}{x_{max} - x_{min}}(r_{max} - r_{min}) + r_{min}$

4) Standardization: gives data the property of a standard normal distribution.

$x' = \frac{x - \mu}{\sigma}$, μ: *mean*, σ: *standard diviation*

The set of new data (X′) has zero mean and unit variance, but not bounded.

Stochastic Gradient Decent Method

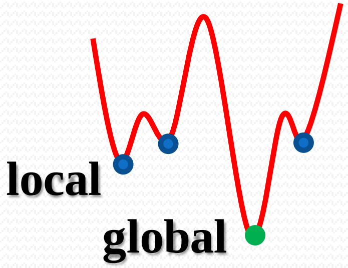
1 - Gradient decent (GD) method ((Batch gradient descent):

$\mathbf{w} = \mathbf{w} + \Delta w$

$\Delta w = -\eta \, \Delta J(w)$

$\Delta wj = -\eta \frac{\partial J}{\partial w_j} = \eta \sum_i \left(y^i - \phi(z^i)\right) x^i{}_{\mathrm{j}}$ Based on all samples

i: Sample index, j: Feature index



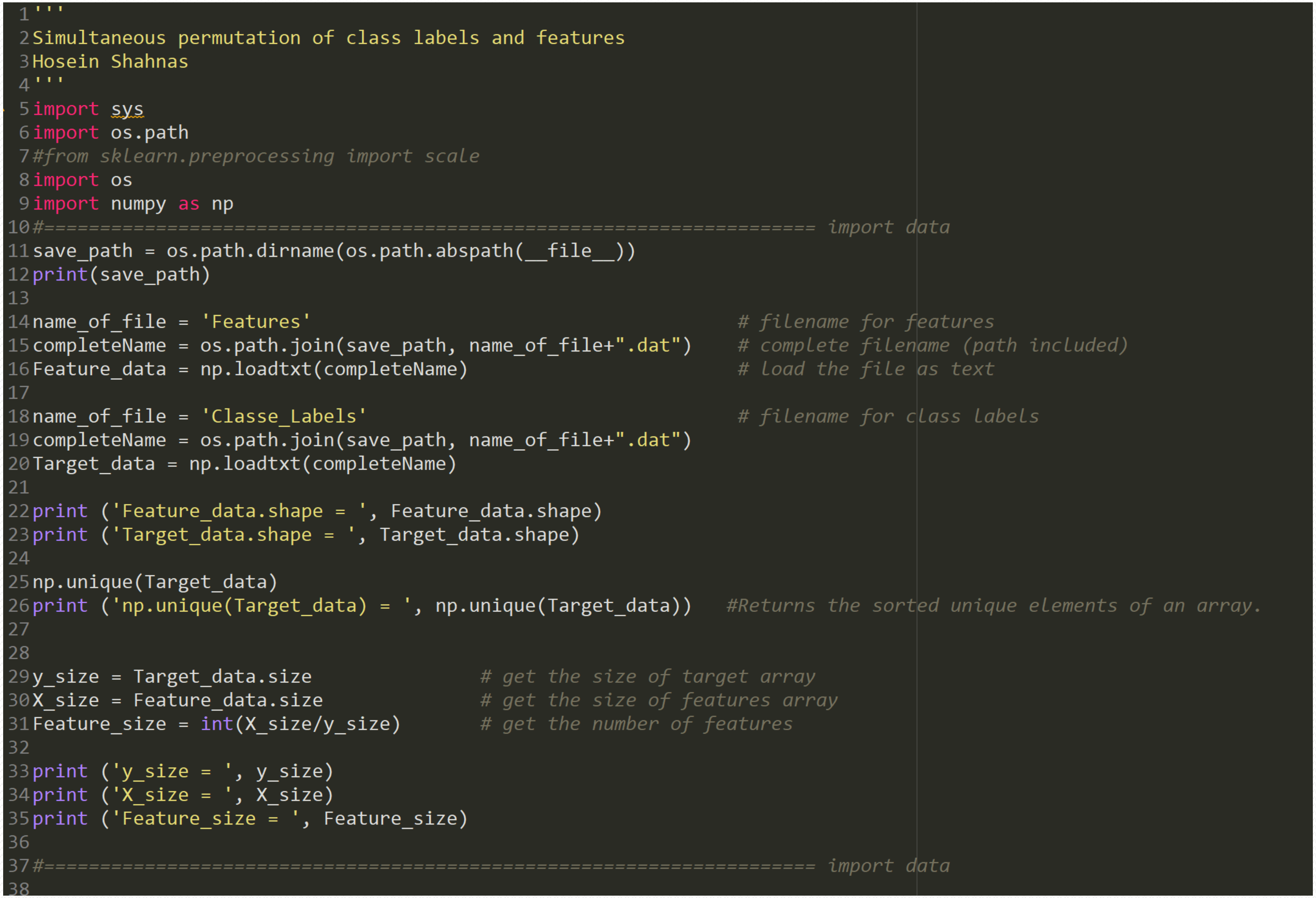
2 – Stochastic gradient decent (SGD) method (an approx. for GD for large data):

$\Delta wj = -\eta \frac{\partial J}{\partial w_j} = \eta \left(y^i - \phi(z^i)\right) x^i{}_{\mathrm{j}}$ Based on random samples

η :Variable learning rate (decreasing with iteration)

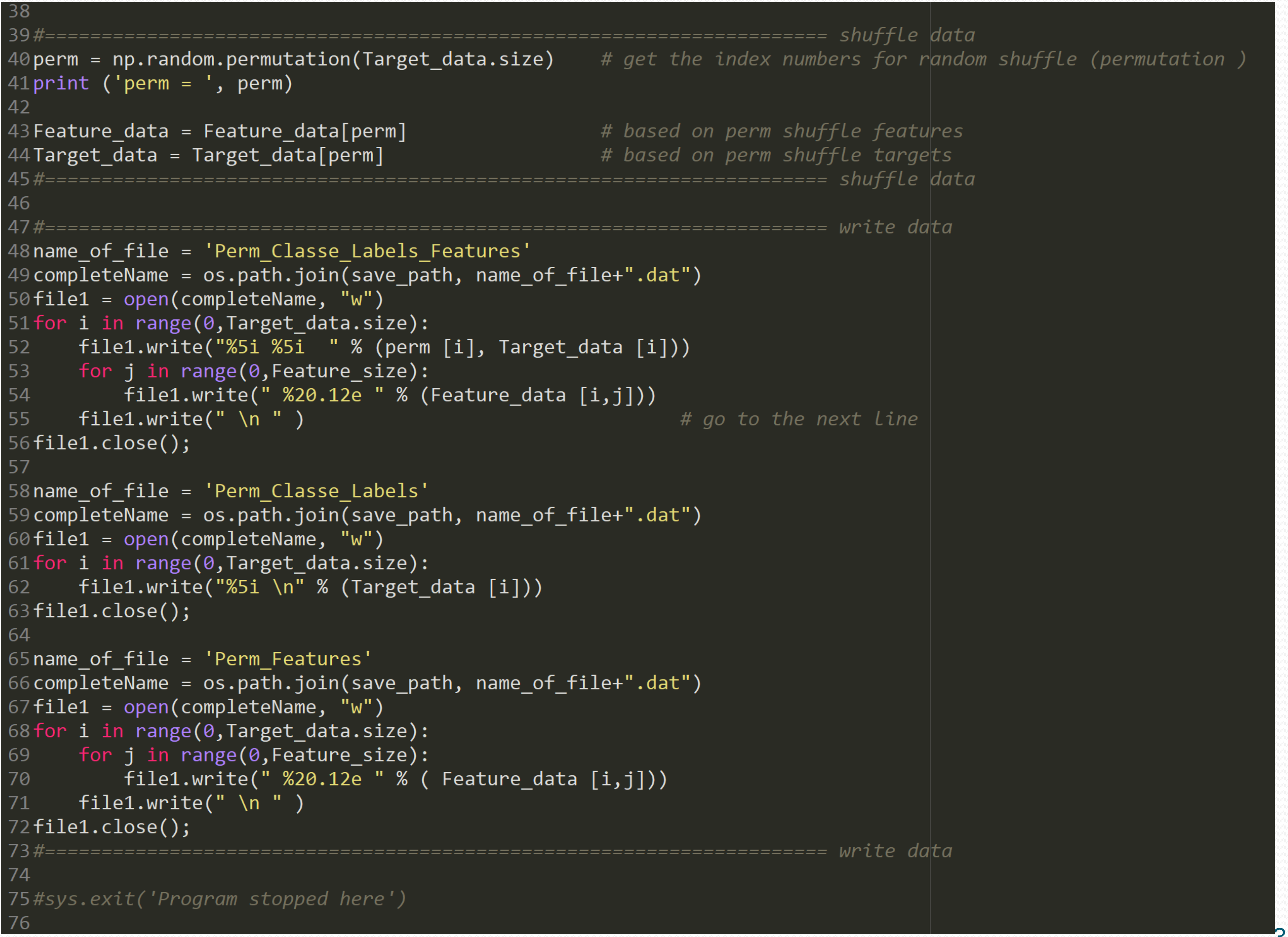
In SGD **a)** the error is noisier than in GD (because $\Delta w's$ are based on single samples), **b)** The convergence is faster than GD (more frequent updates), **c)** the local minima can be escaped faster which is good, **d)** to get accurate results the samples must be shuffled,

Permutation



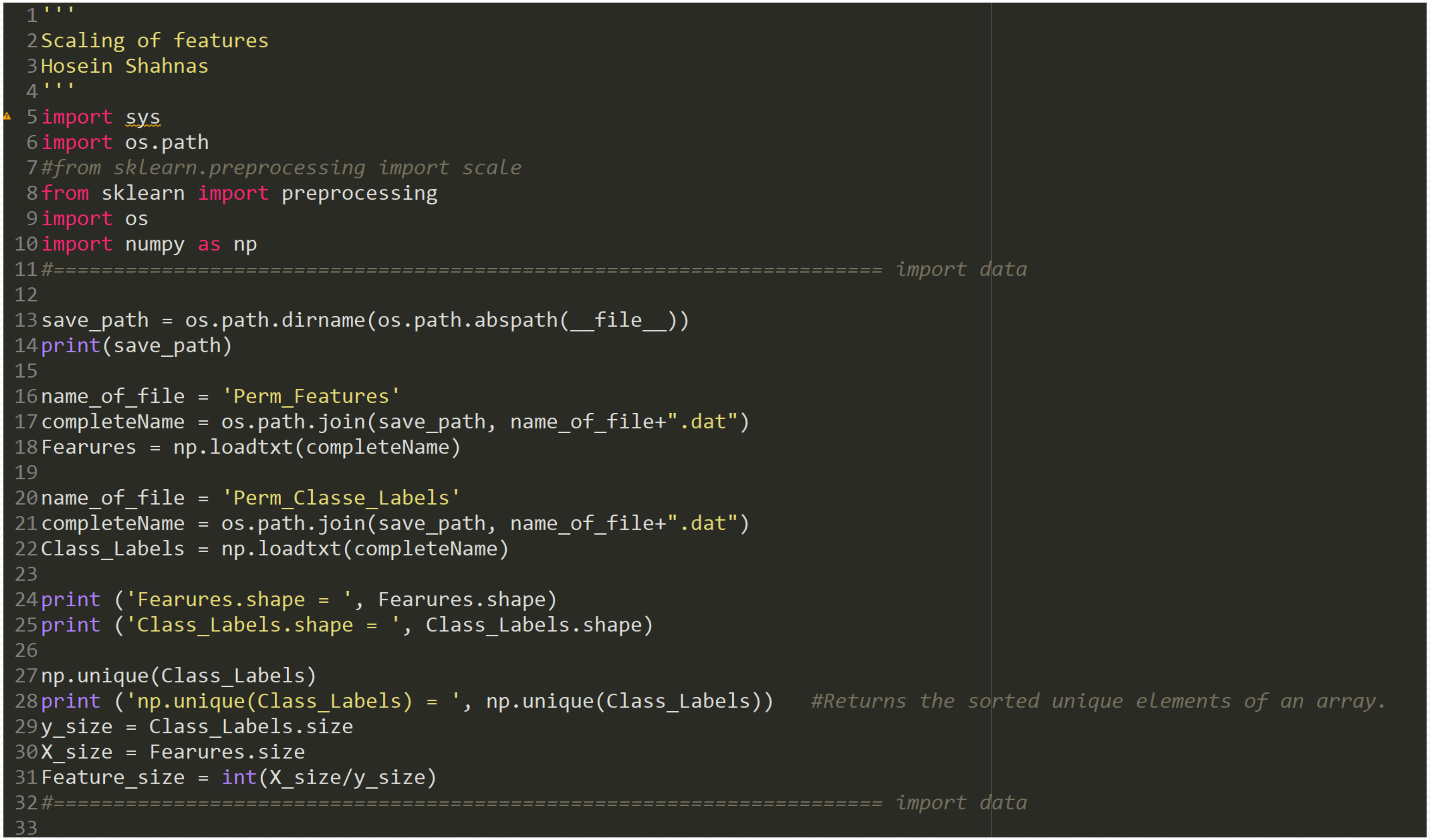
```
'''
Simultaneous permutation of class labels and features
Hosein Shahnas
'''
import sys
import os.path
#from sklearn.preprocessing import scale
import os
import numpy as np
#===================================================================== import data
save_path = os.path.dirname(os.path.abspath(__file__))
print(save_path)

name_of_file = 'Features'                                      # filename for features
completeName = os.path.join(save_path, name_of_file+".dat")    # complete filename (path included)
Feature_data = np.loadtxt(completeName)                        # load the file as text

name_of_file = 'Classe_Labels'                                 # filename for class labels
completeName = os.path.join(save_path, name_of_file+".dat")
Target_data = np.loadtxt(completeName)

print ('Feature_data.shape = ', Feature_data.shape)
print ('Target_data.shape = ', Target_data.shape)

np.unique(Target_data)
print ('np.unique(Target_data) = ', np.unique(Target_data))   #Returns the sorted unique elements of an array.


y_size = Target_data.size               # get the size of target array
X_size = Feature_data.size              # get the size of features array
Feature_size = int(X_size/y_size)       # get the number of features

print ('y_size = ', y_size)
print ('X_size = ', X_size)
print ('Feature_size = ', Feature_size)

#===================================================================== import data
```

Permutation



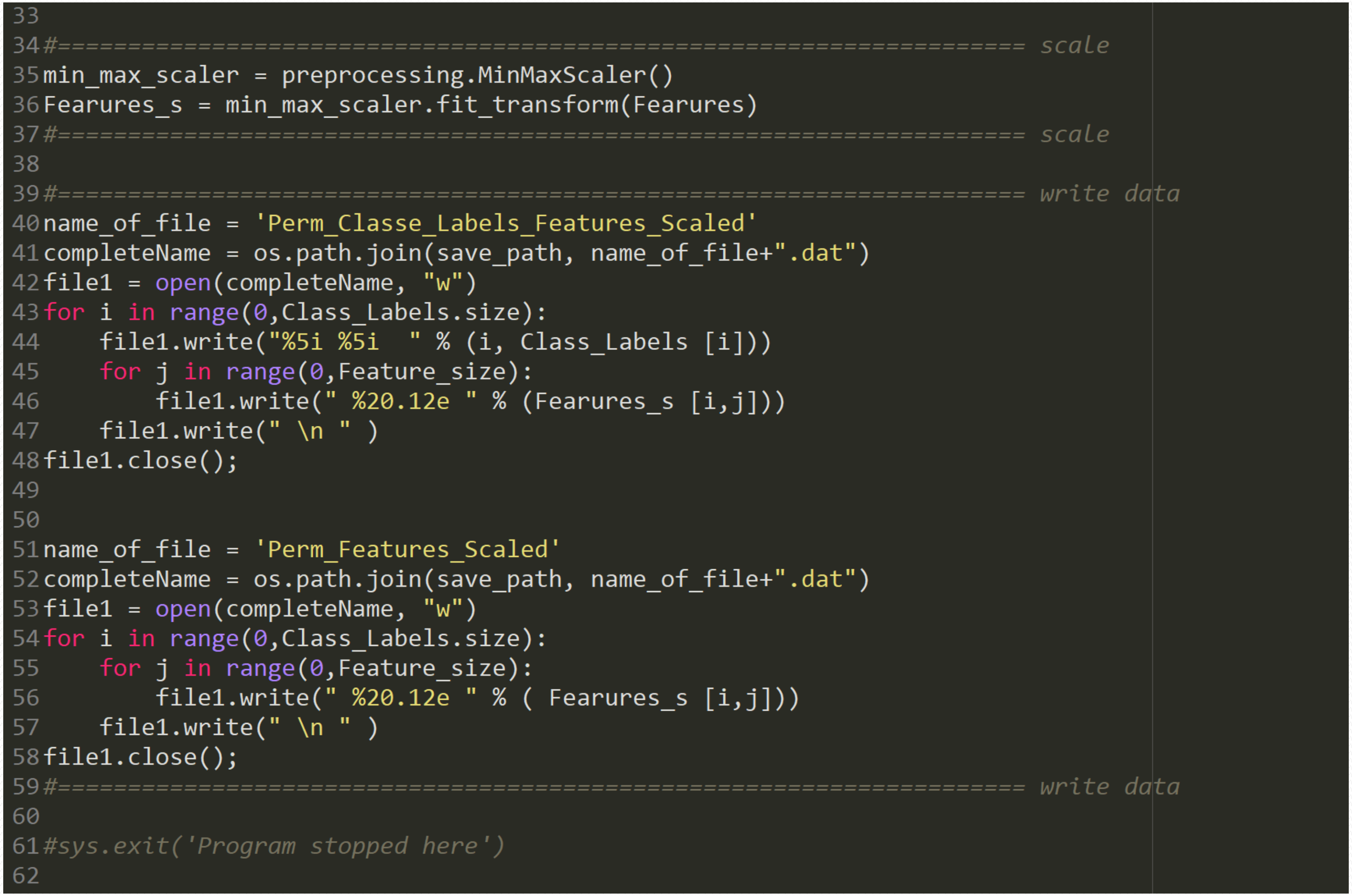
```
#===================================================================== shuffle data
perm = np.random.permutation(Target_data.size)    # get the index numbers for random shuffle (permutation )
print ('perm = ', perm)

Feature_data = Feature_data[perm]                 # based on perm shuffle features
Target_data = Target_data[perm]                   # based on perm shuffle targets
#===================================================================== shuffle data

#===================================================================== write data
name_of_file = 'Perm_Classe_Labels_Features'
completeName = os.path.join(save_path, name_of_file+".dat")
file1 = open(completeName, "w")
for i in range(0,Target_data.size):
    file1.write("%5i %5i  " % (perm [i], Target_data [i]))
    for j in range(0,Feature_size):
        file1.write(" %20.12e " % (Feature_data [i,j]))
    file1.write(" \n " )                                 # go to the next line
file1.close();

name_of_file = 'Perm_Classe_Labels'
completeName = os.path.join(save_path, name_of_file+".dat")
file1 = open(completeName, "w")
for i in range(0,Target_data.size):
    file1.write("%5i \n" % (Target_data [i]))
file1.close();

name_of_file = 'Perm_Features'
completeName = os.path.join(save_path, name_of_file+".dat")
file1 = open(completeName, "w")
for i in range(0,Target_data.size):
    for j in range(0,Feature_size):
        file1.write(" %20.12e " % ( Feature_data [i,j]))
    file1.write(" \n " )
file1.close();
#===================================================================== write data

#sys.exit('Program stopped here')
```

Scaling



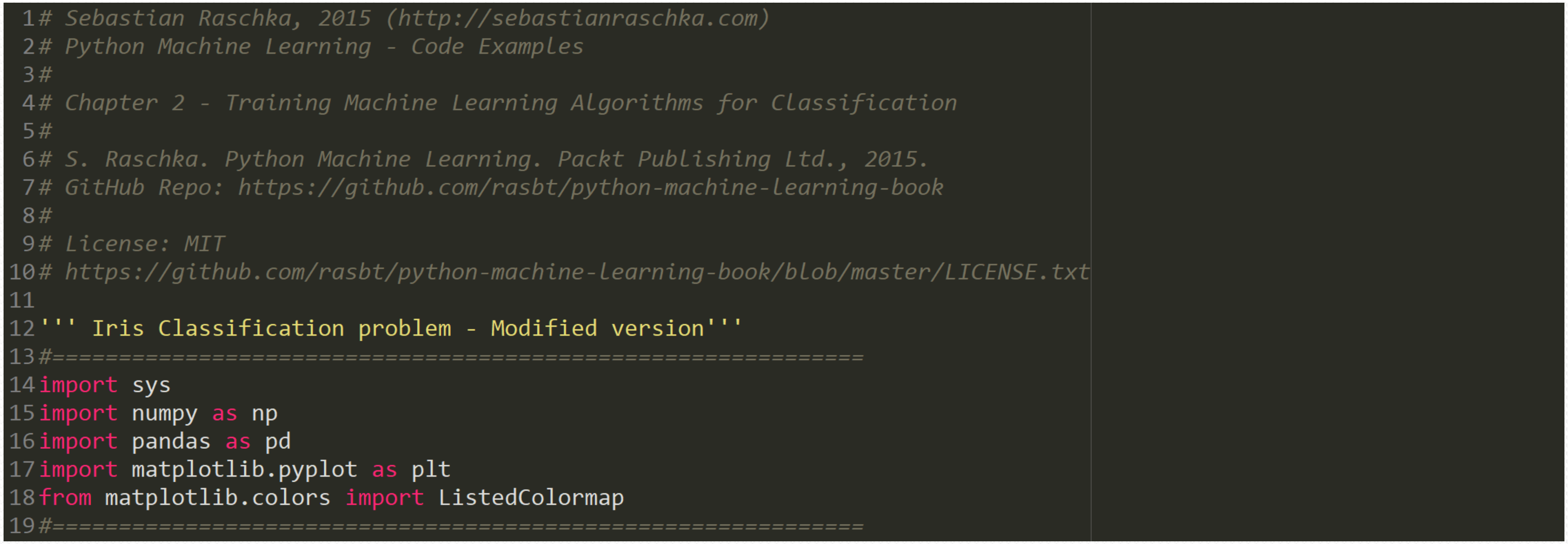
```
'''
Scaling of features
Hosein Shahnas
'''
import sys
import os.path
#from sklearn.preprocessing import scale
from sklearn import preprocessing
import os
import numpy as np
#===================================================================== import data

save_path = os.path.dirname(os.path.abspath(__file__))
print(save_path)

name_of_file = 'Perm_Features'
completeName = os.path.join(save_path, name_of_file+".dat")
Fearures = np.loadtxt(completeName)

name_of_file = 'Perm_Classe_Labels'
completeName = os.path.join(save_path, name_of_file+".dat")
Class_Labels = np.loadtxt(completeName)

print ('Fearures.shape = ', Fearures.shape)
print ('Class_Labels.shape = ', Class_Labels.shape)

np.unique(Class_Labels)
print ('np.unique(Class_Labels) = ', np.unique(Class_Labels))   #Returns the sorted unique elements of an array.
y_size = Class_Labels.size
X_size = Fearures.size
Feature_size = int(X_size/y_size)
#===================================================================== import data

```

Scaling



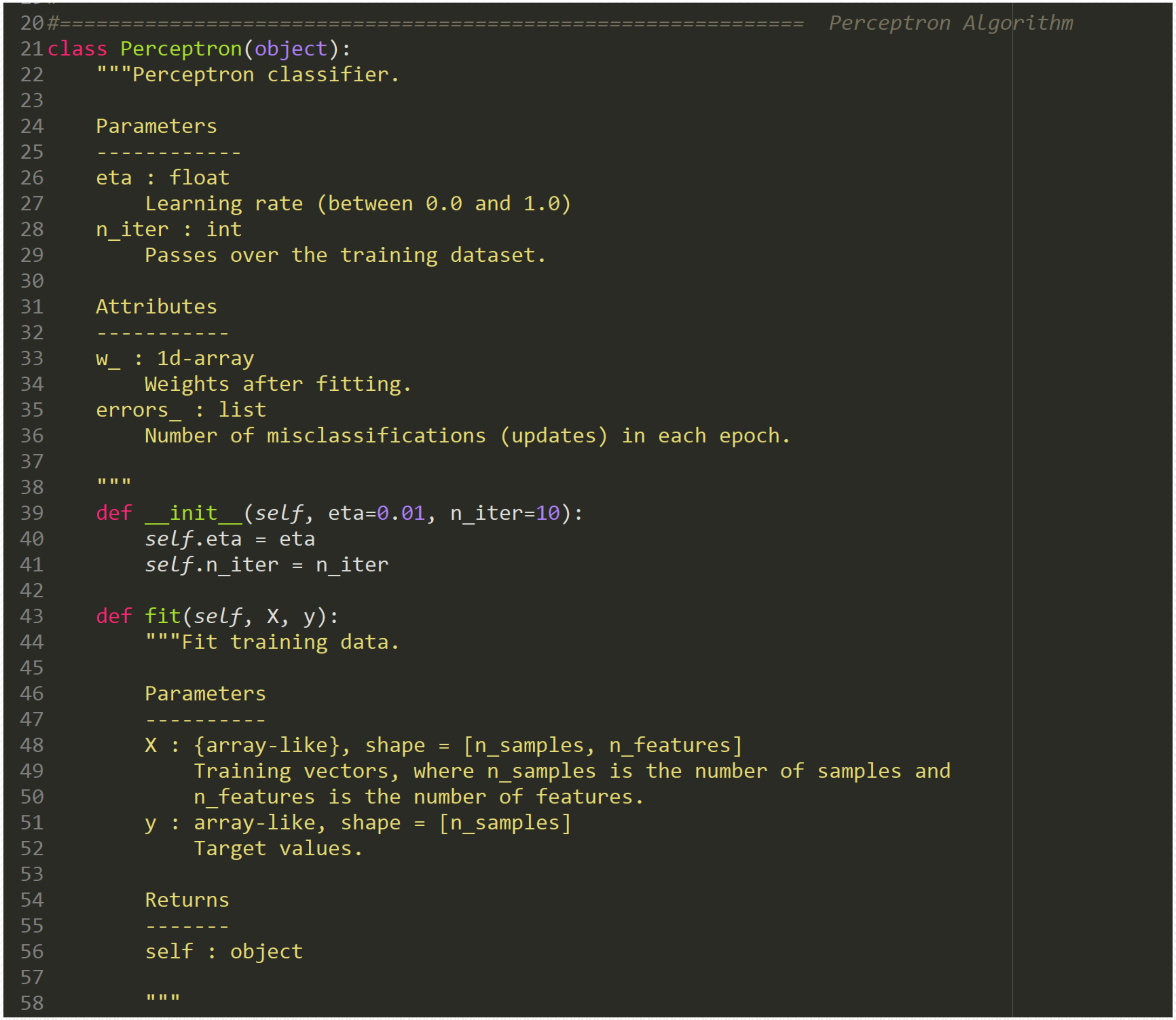
```

#===================================================================== scale
min_max_scaler = preprocessing.MinMaxScaler()
Fearures_s = min_max_scaler.fit_transform(Fearures)
#===================================================================== scale

#===================================================================== write data
name_of_file = 'Perm_Classe_Labels_Features_Scaled'
completeName = os.path.join(save_path, name_of_file+".dat")
file1 = open(completeName, "w")
for i in range(0,Class_Labels.size):
    file1.write("%5i %5i  " % (i, Class_Labels [i]))
    for j in range(0,Feature_size):
        file1.write(" %20.12e " % (Fearures_s [i,j]))
    file1.write(" \n " )
file1.close();


name_of_file = 'Perm_Features_Scaled'
completeName = os.path.join(save_path, name_of_file+".dat")
file1 = open(completeName, "w")
for i in range(0,Class_Labels.size):
    for j in range(0,Feature_size):
        file1.write(" %20.12e " % ( Fearures_s [i,j]))
    file1.write(" \n " )
file1.close();
#===================================================================== write data

#sys.exit('Program stopped here')

```

Perceptron, Adaline-GD, Adaline-SGD Algorithms



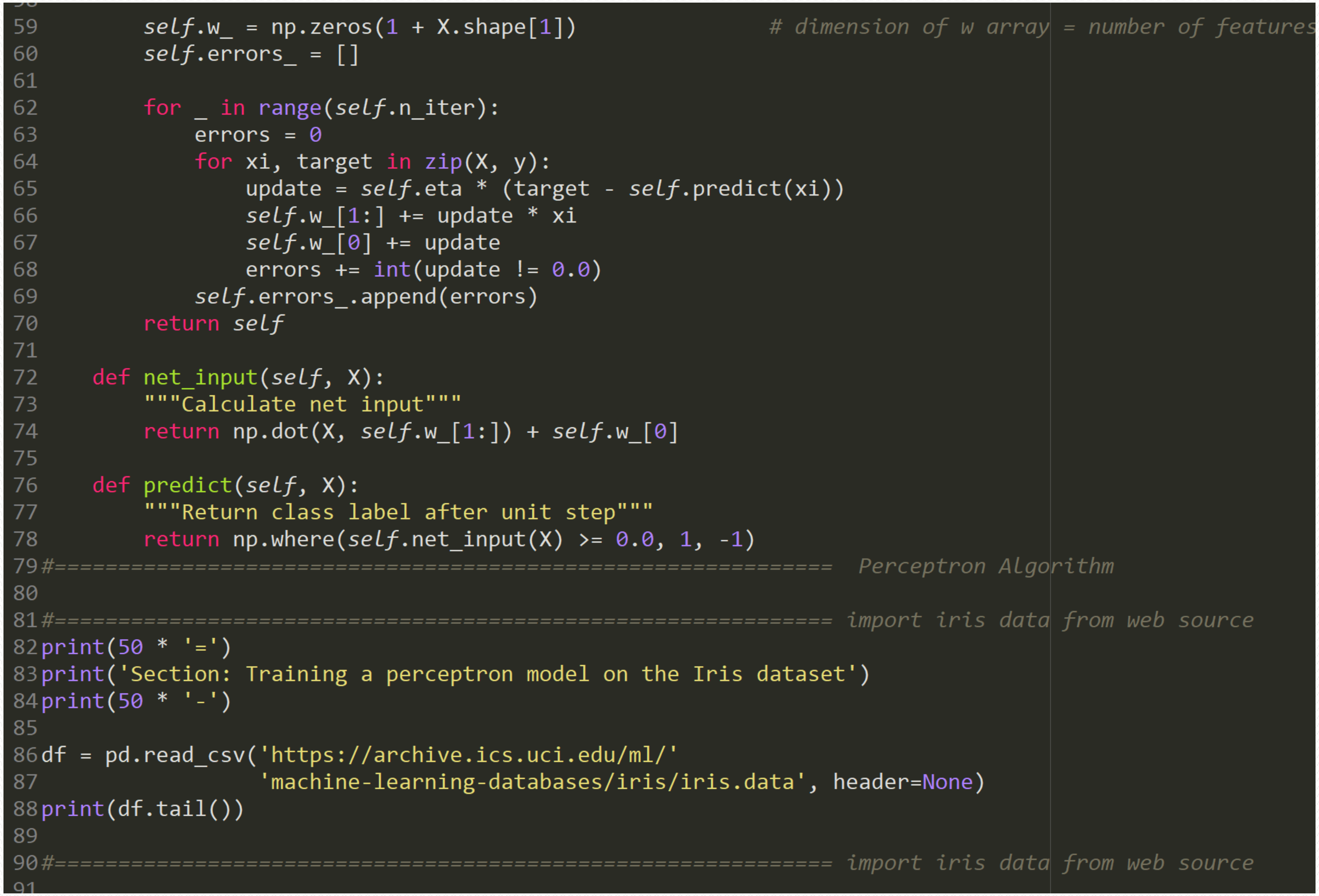
```python
# Sebastian Raschka, 2015 (http://sebastianraschka.com)
# Python Machine Learning - Code Examples
#
# Chapter 2 - Training Machine Learning Algorithms for Classification
#
# S. Raschka. Python Machine Learning. Packt Publishing Ltd., 2015.
# GitHub Repo: https://github.com/rasbt/python-machine-learning-book
#
# License: MIT
# https://github.com/rasbt/python-machine-learning-book/blob/master/LICENSE.txt

''' Iris Classification problem - Modified version'''
#==============================================================
import sys
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
#==============================================================
```

Perceptron, Adaline-GD, Adaline-SGD Algorithms



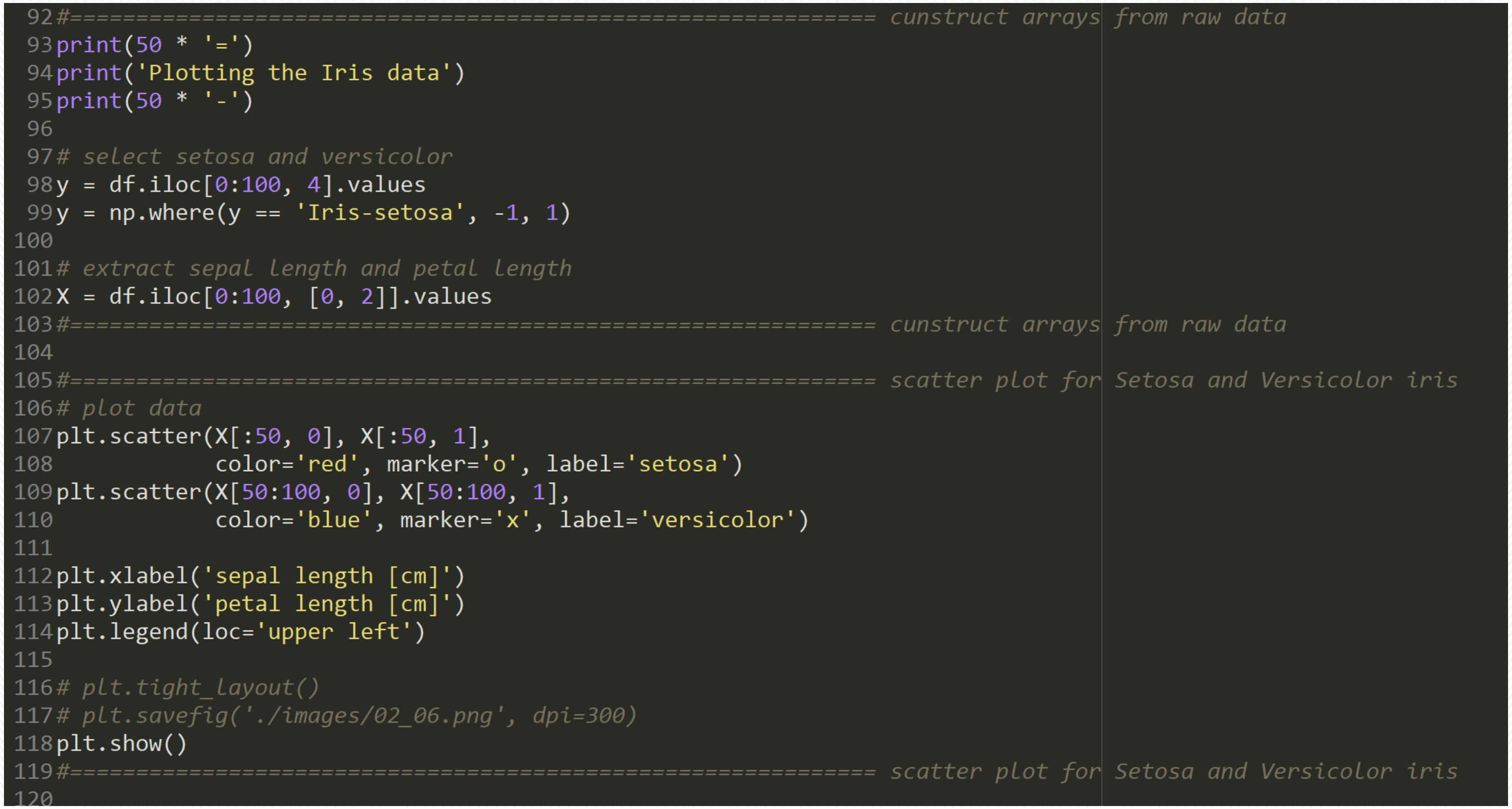
```python
#=============================================================  Perceptron Algorithm
class Perceptron(object):
    """Perceptron classifier.

    Parameters
    ------------
    eta : float
        Learning rate (between 0.0 and 1.0)
    n_iter : int
        Passes over the training dataset.

    Attributes
    -----------
    w_ : 1d-array
        Weights after fitting.
    errors_ : list
        Number of misclassifications (updates) in each epoch.

    """
    def __init__(self, eta=0.01, n_iter=10):
        self.eta = eta
        self.n_iter = n_iter

    def fit(self, X, y):
        """Fit training data.

        Parameters
        ----------
        X : {array-like}, shape = [n_samples, n_features]
            Training vectors, where n_samples is the number of samples and
            n_features is the number of features.
        y : array-like, shape = [n_samples]
            Target values.

        Returns
        -------
        self : object

        """
```

Perceptron, Adaline-GD, Adaline-SGD Algorithms



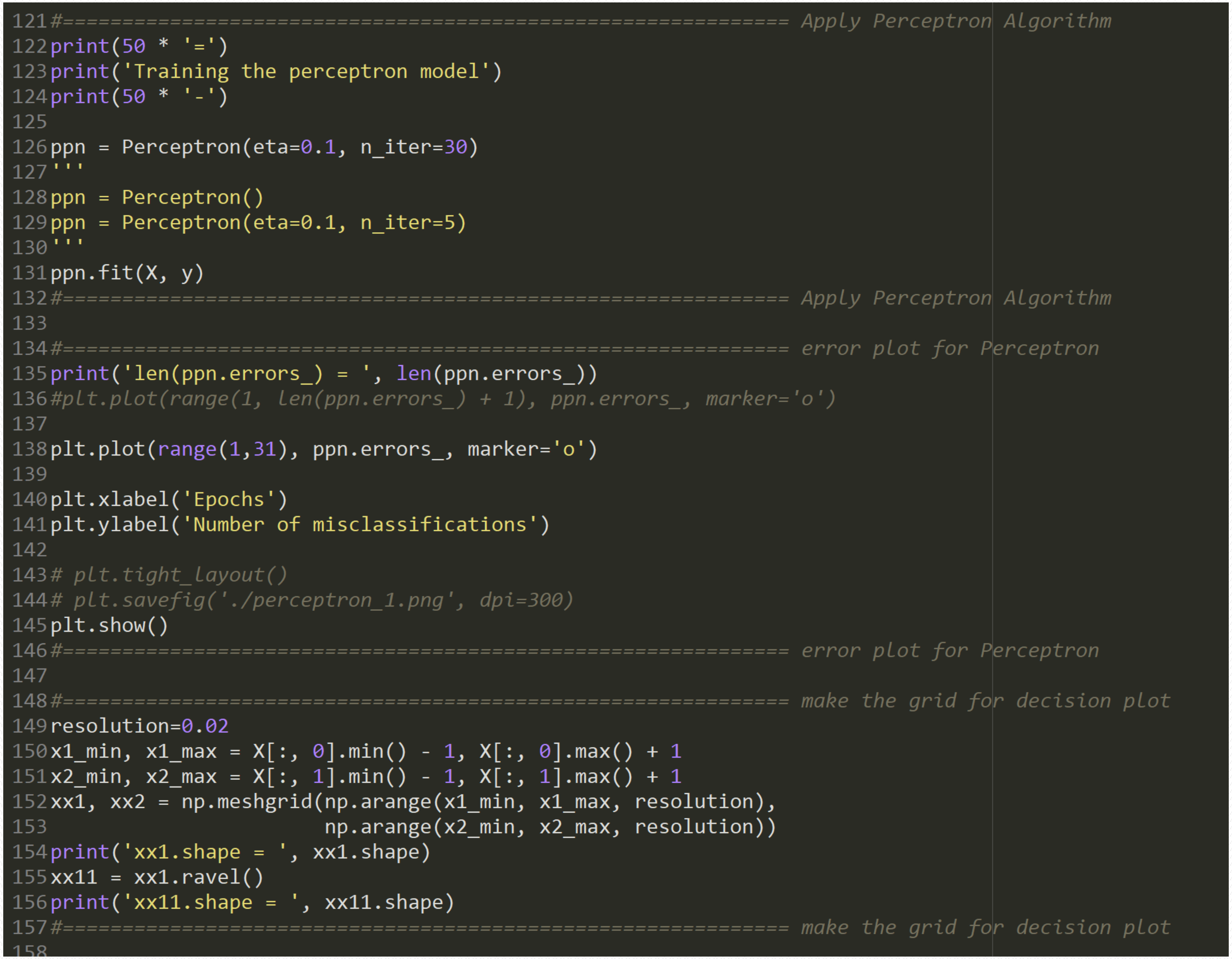
```
        self.w_ = np.zeros(1 + X.shape[1])               # dimension of w array = number of features
        self.errors_ = []

        for _ in range(self.n_iter):
            errors = 0
            for xi, target in zip(X, y):
                update = self.eta * (target - self.predict(xi))
                self.w_[1:] += update * xi
                self.w_[0] += update
                errors += int(update != 0.0)
            self.errors_.append(errors)
        return self

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

    def predict(self, X):
        """Return class label after unit step"""
        return np.where(self.net_input(X) >= 0.0, 1, -1)
#==============================================================  Perceptron Algorithm

#============================================================== import iris data from web source
print(50 * '=')
print('Section: Training a perceptron model on the Iris dataset')
print(50 * '-')

df = pd.read_csv('https://archive.ics.uci.edu/ml/'
                 'machine-learning-databases/iris/iris.data', header=None)
print(df.tail())

#============================================================== import iris data from web source
```

Perceptron, Adaline-GD, Adaline-SGD Algorithms



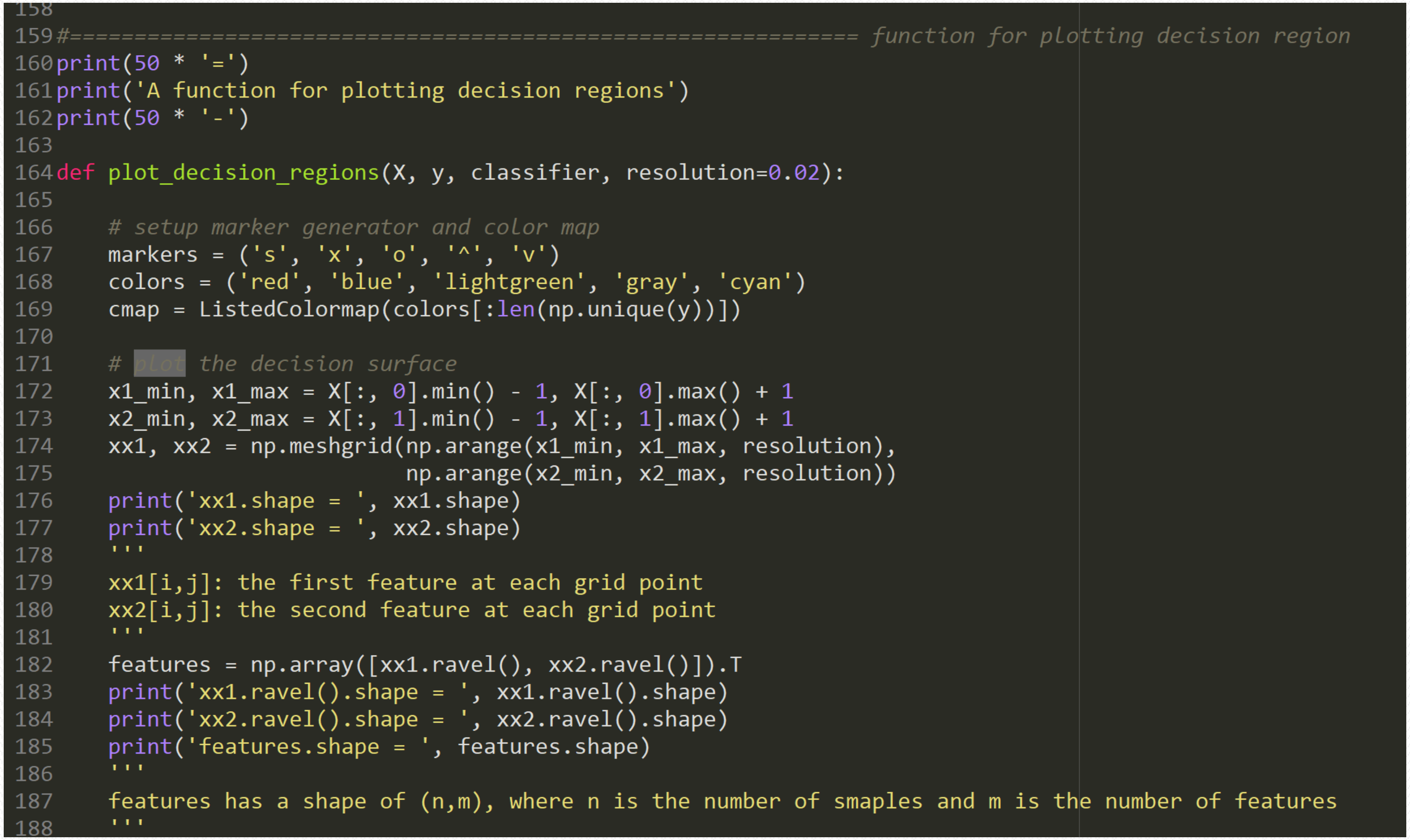
```python
#============================================================= cunstruct arrays from raw data
print(50 * '=')
print('Plotting the Iris data')
print(50 * '-')

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

# extract sepal length and petal length
X = df.iloc[0:100, [0, 2]].values
#============================================================= cunstruct arrays from raw data

#============================================================= scatter plot for Setosa and Versicolor iris
# plot data
plt.scatter(X[:50, 0], X[:50, 1],
            color='red', marker='o', label='setosa')
plt.scatter(X[50:100, 0], X[50:100, 1],
            color='blue', marker='x', label='versicolor')

plt.xlabel('sepal length [cm]')
plt.ylabel('petal length [cm]')
plt.legend(loc='upper left')

# plt.tight_layout()
# plt.savefig('./images/02_06.png', dpi=300)
plt.show()
#============================================================= scatter plot for Setosa and Versicolor iris

```

Perceptron, Adaline-GD, Adaline-SGD Algorithms



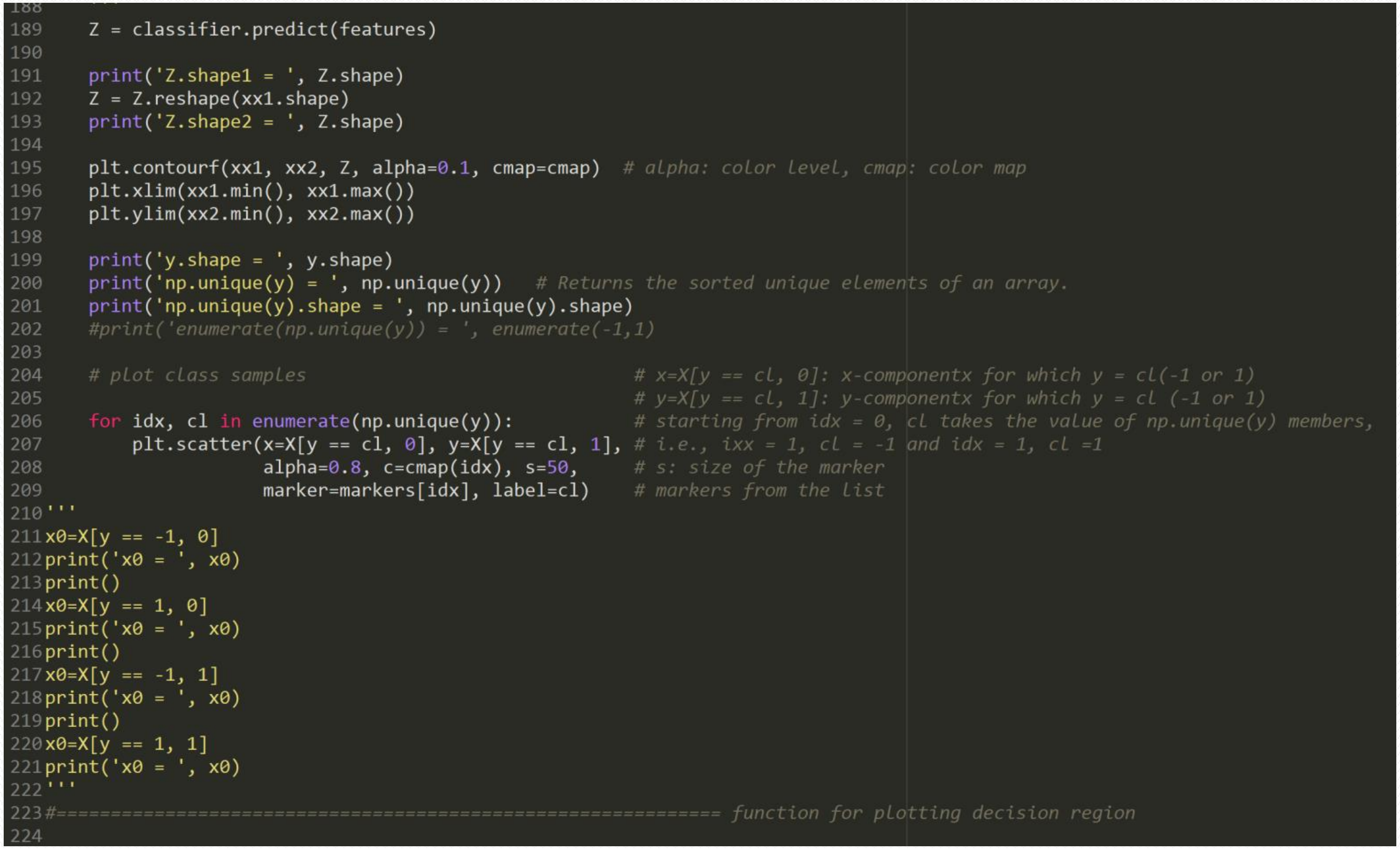
```
#============================================================= Apply Perceptron Algorithm
print(50 * '=')
print('Training the perceptron model')
print(50 * '-')

ppn = Perceptron(eta=0.1, n_iter=30)
'''
ppn = Perceptron()
ppn = Perceptron(eta=0.1, n_iter=5)
'''
ppn.fit(X, y)
#============================================================= Apply Perceptron Algorithm

#============================================================= error plot for Perceptron
print('len(ppn.errors_) = ', len(ppn.errors_))
#plt.plot(range(1, len(ppn.errors_) + 1), ppn.errors_, marker='o')

plt.plot(range(1,31), ppn.errors_, marker='o')

plt.xlabel('Epochs')
plt.ylabel('Number of misclassifications')

# plt.tight_layout()
# plt.savefig('./perceptron_1.png', dpi=300)
plt.show()
#============================================================= error plot for Perceptron

#============================================================= make the grid for decision plot
resolution=0.02
x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution),
                       np.arange(x2_min, x2_max, resolution))
print('xx1.shape = ', xx1.shape)
xx11 = xx1.ravel()
print('xx11.shape = ', xx11.shape)
#============================================================= make the grid for decision plot
```

Perceptron, Adaline-GD, Adaline-SGD Algorithms



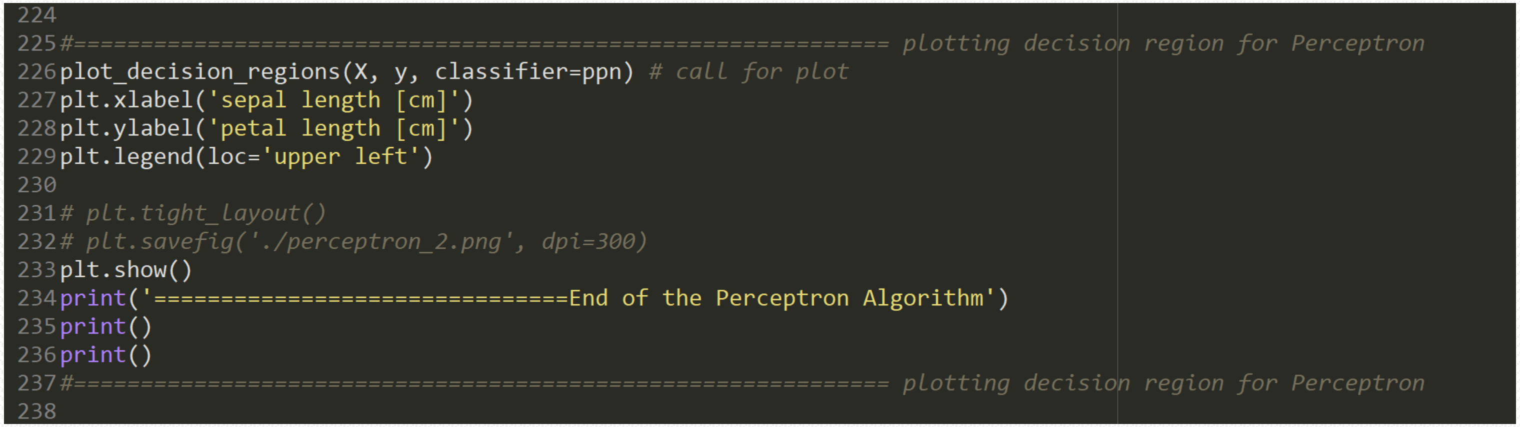
```
#============================================================== function for plotting decision region
print(50 * '=')
print('A function for plotting decision regions')
print(50 * '-')

def plot_decision_regions(X, y, classifier, resolution=0.02):

    # setup marker generator and color map
    markers = ('s', 'x', 'o', '^', 'v')
    colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
    cmap = ListedColormap(colors[:len(np.unique(y))])

    # plot the decision surface
    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution),
                           np.arange(x2_min, x2_max, resolution))
    print('xx1.shape = ', xx1.shape)
    print('xx2.shape = ', xx2.shape)
    '''
    xx1[i,j]: the first feature at each grid point
    xx2[i,j]: the second feature at each grid point
    '''
    features = np.array([xx1.ravel(), xx2.ravel()]).T
    print('xx1.ravel().shape = ', xx1.ravel().shape)
    print('xx2.ravel().shape = ', xx2.ravel().shape)
    print('features.shape = ', features.shape)
    '''
    features has a shape of (n,m), where n is the number of smaples and m is the number of features
    '''
```

Perceptron, Adaline-GD, Adaline-SGD Algorithms



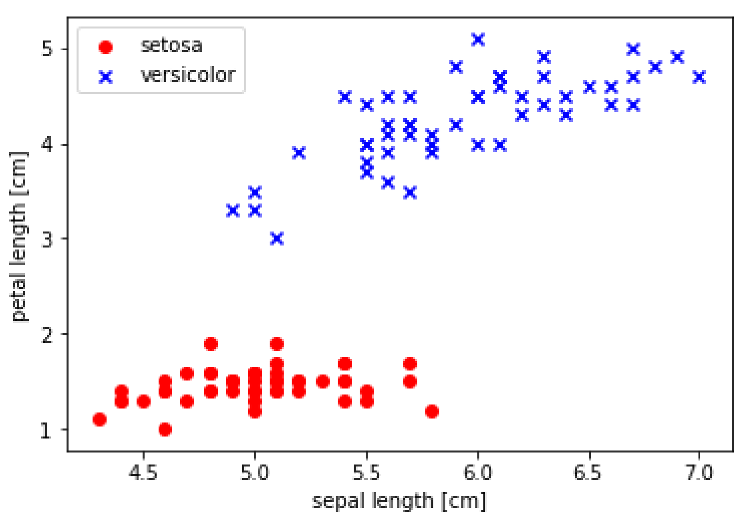
```
    Z = classifier.predict(features)

    print('Z.shape1 = ', Z.shape)
    Z = Z.reshape(xx1.shape)
    print('Z.shape2 = ', Z.shape)

    plt.contourf(xx1, xx2, Z, alpha=0.1, cmap=cmap)  # alpha: color level, cmap: color map
    plt.xlim(xx1.min(), xx1.max())
    plt.ylim(xx2.min(), xx2.max())

    print('y.shape = ', y.shape)
    print('np.unique(y) = ', np.unique(y))   # Returns the sorted unique elements of an array.
    print('np.unique(y).shape = ', np.unique(y).shape)
    #print('enumerate(np.unique(y)) = ', enumerate(-1,1)

    # plot class samples                              # x=X[y == cl, 0]: x-componentx for which y = cl(-1 or 1)
                                                      # y=X[y == cl, 1]: y-componentx for which y = cl (-1 or 1)
    for idx, cl in enumerate(np.unique(y)):           # starting from idx = 0, cl takes the value of np.unique(y) members,
        plt.scatter(x=X[y == cl, 0], y=X[y == cl, 1], # i.e., ixx = 1, cl = -1 and idx = 1, cl =1
                    alpha=0.8, c=cmap(idx), s=50,     # s: size of the marker
                    marker=markers[idx], label=cl)    # markers from the list
'''
x0=X[y == -1, 0]
print('x0 = ', x0)
print()
x0=X[y == 1, 0]
print('x0 = ', x0)
print()
x0=X[y == -1, 1]
print('x0 = ', x0)
print()
x0=X[y == 1, 1]
print('x0 = ', x0)
'''
#============================================================= function for plotting decision region

```

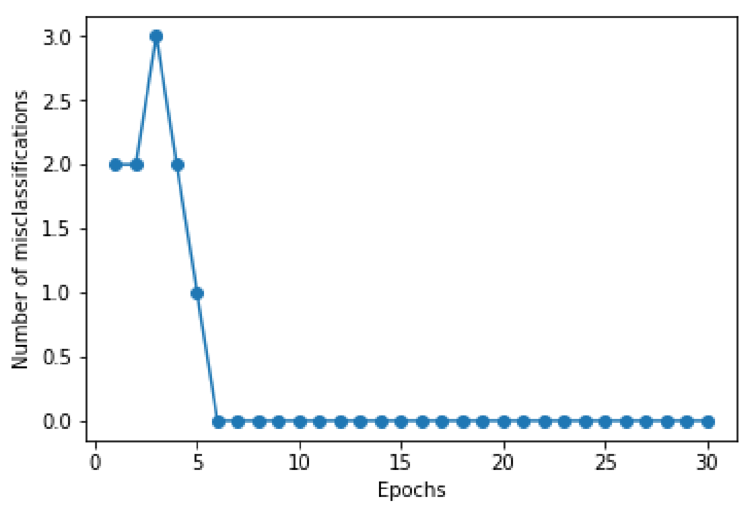
Perceptron, Adaline-GD, Adaline-SGD Algorithms

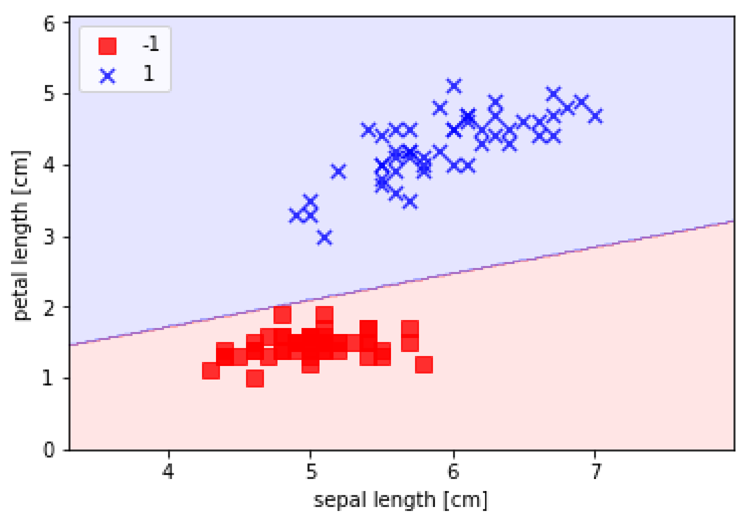


```python

#============================================================= plotting decision region for Perceptron
plot_decision_regions(X, y, classifier=ppn) # call for plot
plt.xlabel('sepal length [cm]')
plt.ylabel('petal length [cm]')
plt.legend(loc='upper left')

# plt.tight_layout()
# plt.savefig('./perceptron_2.png', dpi=300)
plt.show()
print('===============================End of the Perceptron Algorithm')
print()
print()
#============================================================= plotting decision region for Perceptron

```

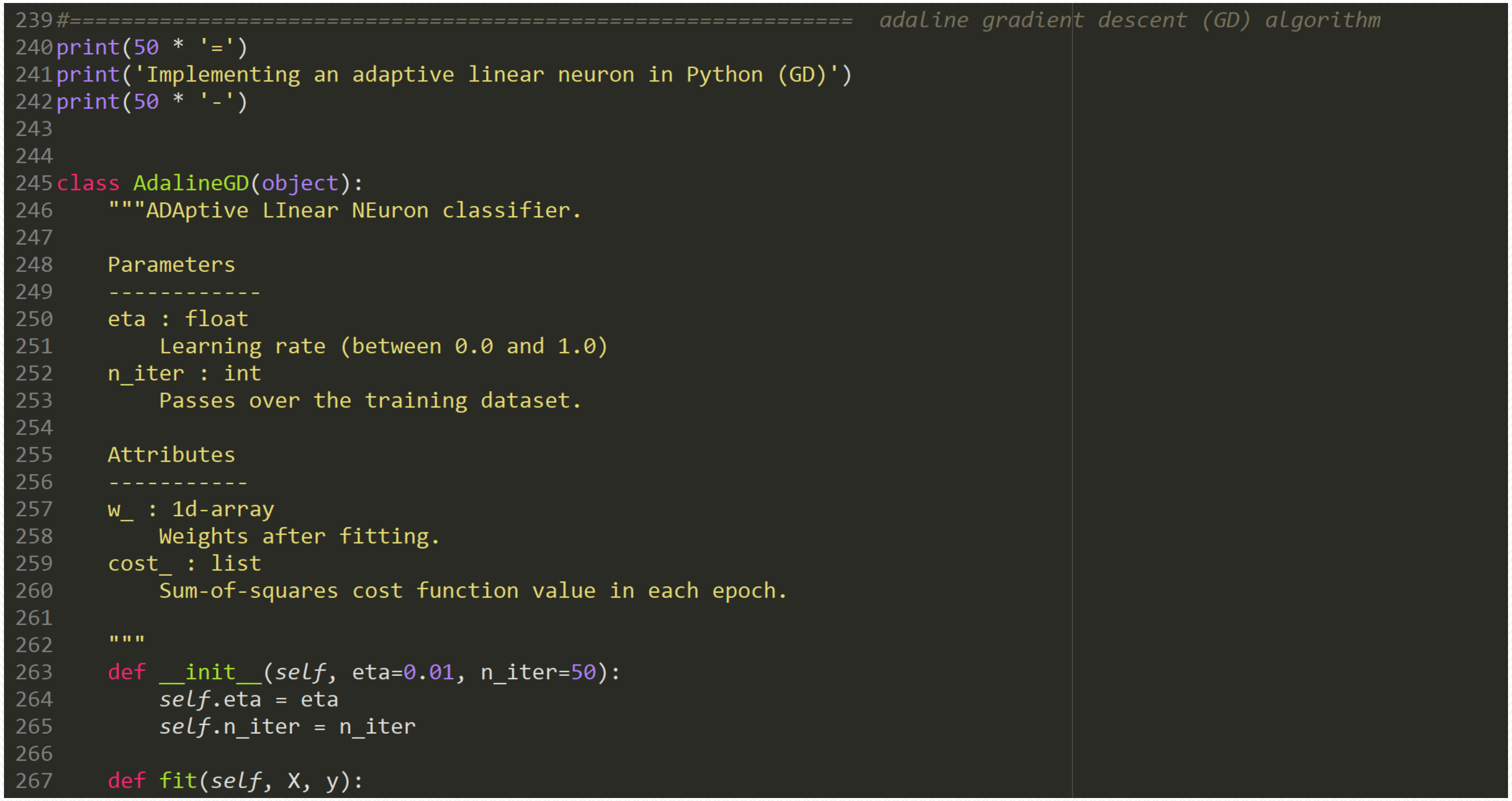
Perceptron


setosa
versicolor
petal length [cm]
5
4
3
2
1
4.5
5.0
5.5
6.0
6.5
7.0
sepal length [cm]


Number of misclassifications
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0
5
10
15
20
25
30
Epochs

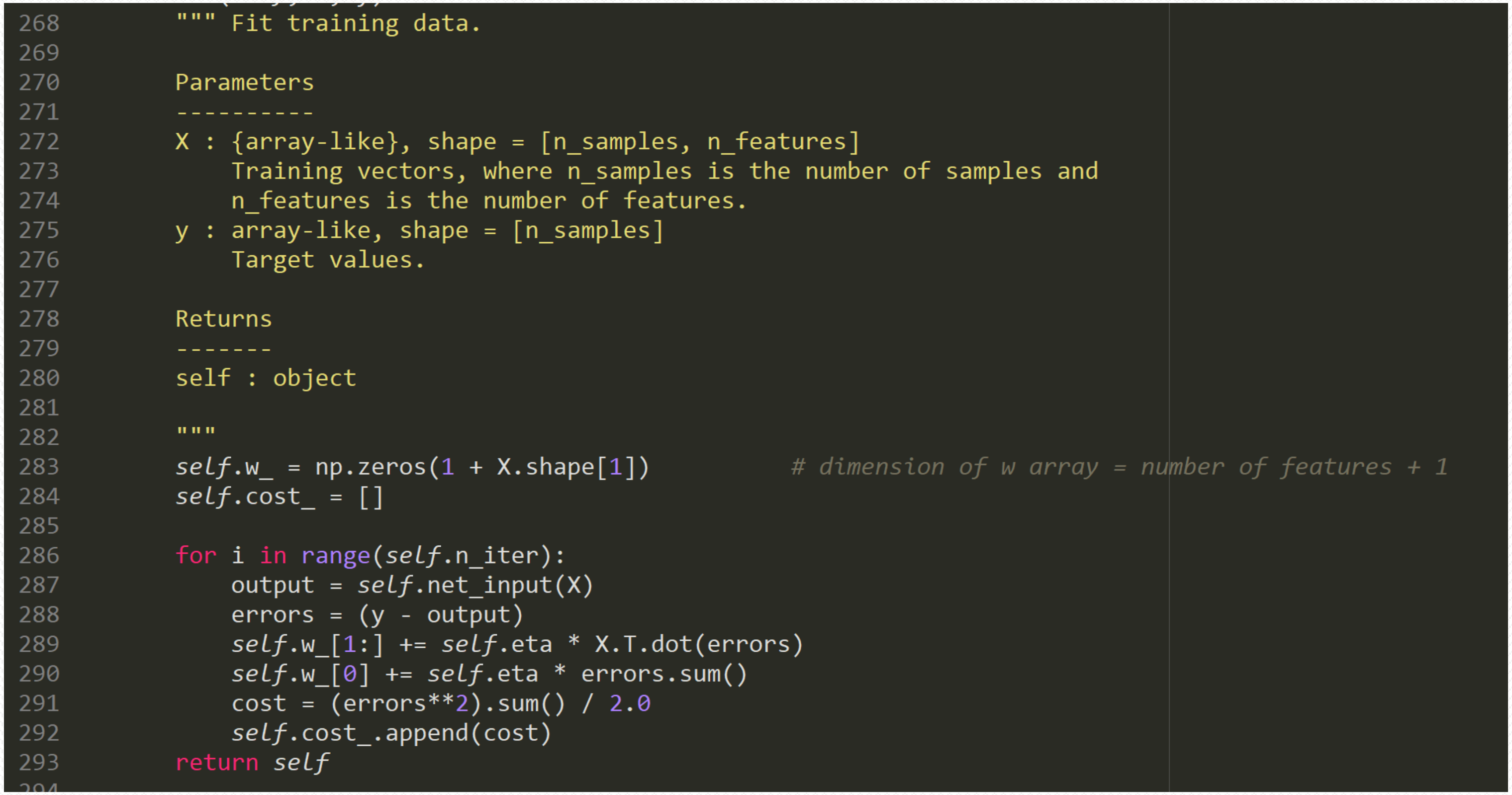

-1
1
petal length [cm]
6
5
4
3
2
1
0
4
5
6
7
sepal length [cm]

Perceptron, Adaline-GD, Adaline-SGD Algorithms



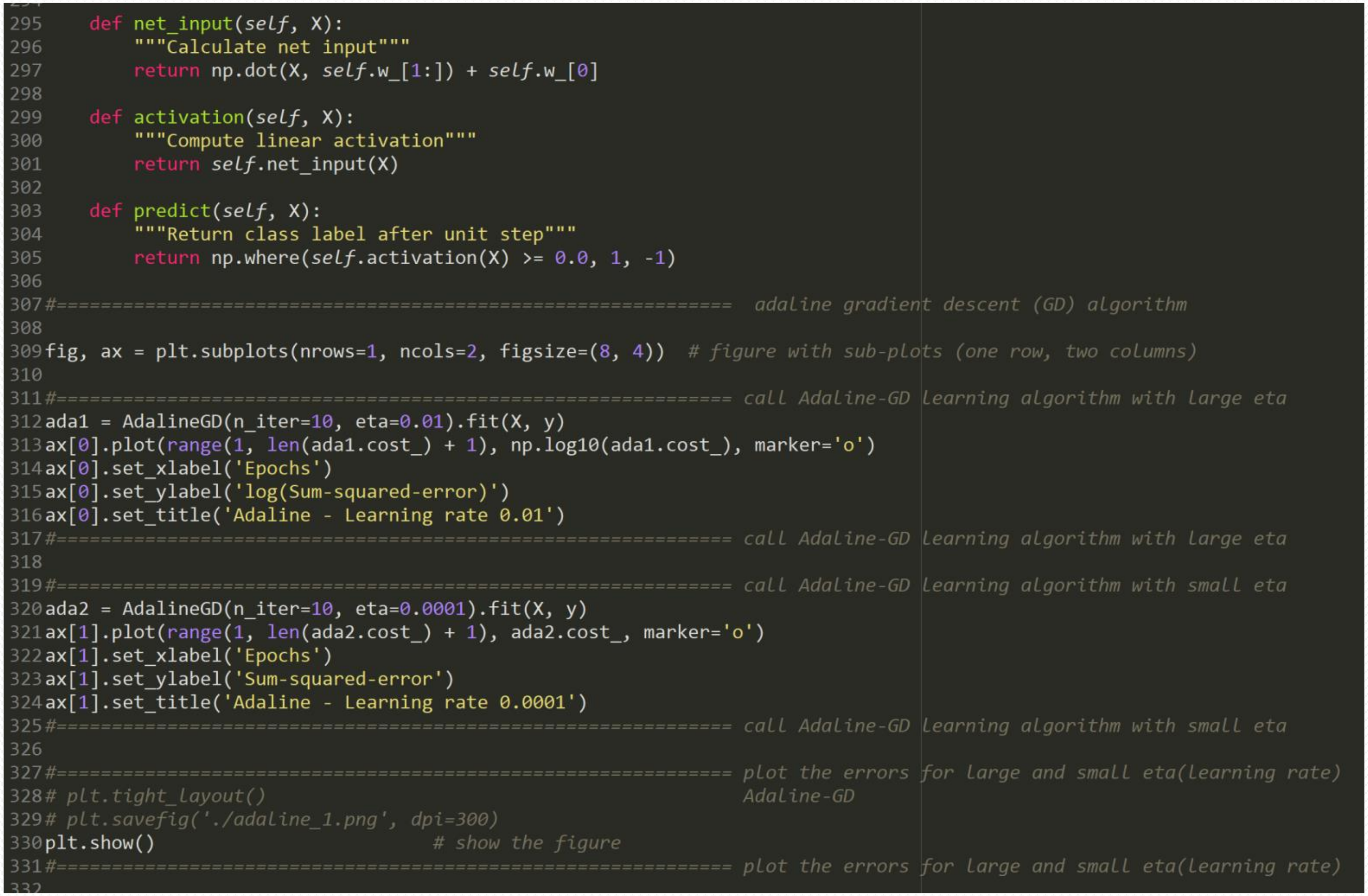
```
#=============================================================  adaline gradient descent (GD) algorithm
print(50 * '=')
print('Implementing an adaptive linear neuron in Python (GD)')
print(50 * '-')


class AdalineGD(object):
    """ADAptive LInear NEuron classifier.

    Parameters
    ------------
    eta : float
        Learning rate (between 0.0 and 1.0)
    n_iter : int
        Passes over the training dataset.

    Attributes
    -----------
    w_ : 1d-array
        Weights after fitting.
    cost_ : list
        Sum-of-squares cost function value in each epoch.

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

    def fit(self, X, y):
```

Perceptron, Adaline-GD, Adaline-SGD Algorithms



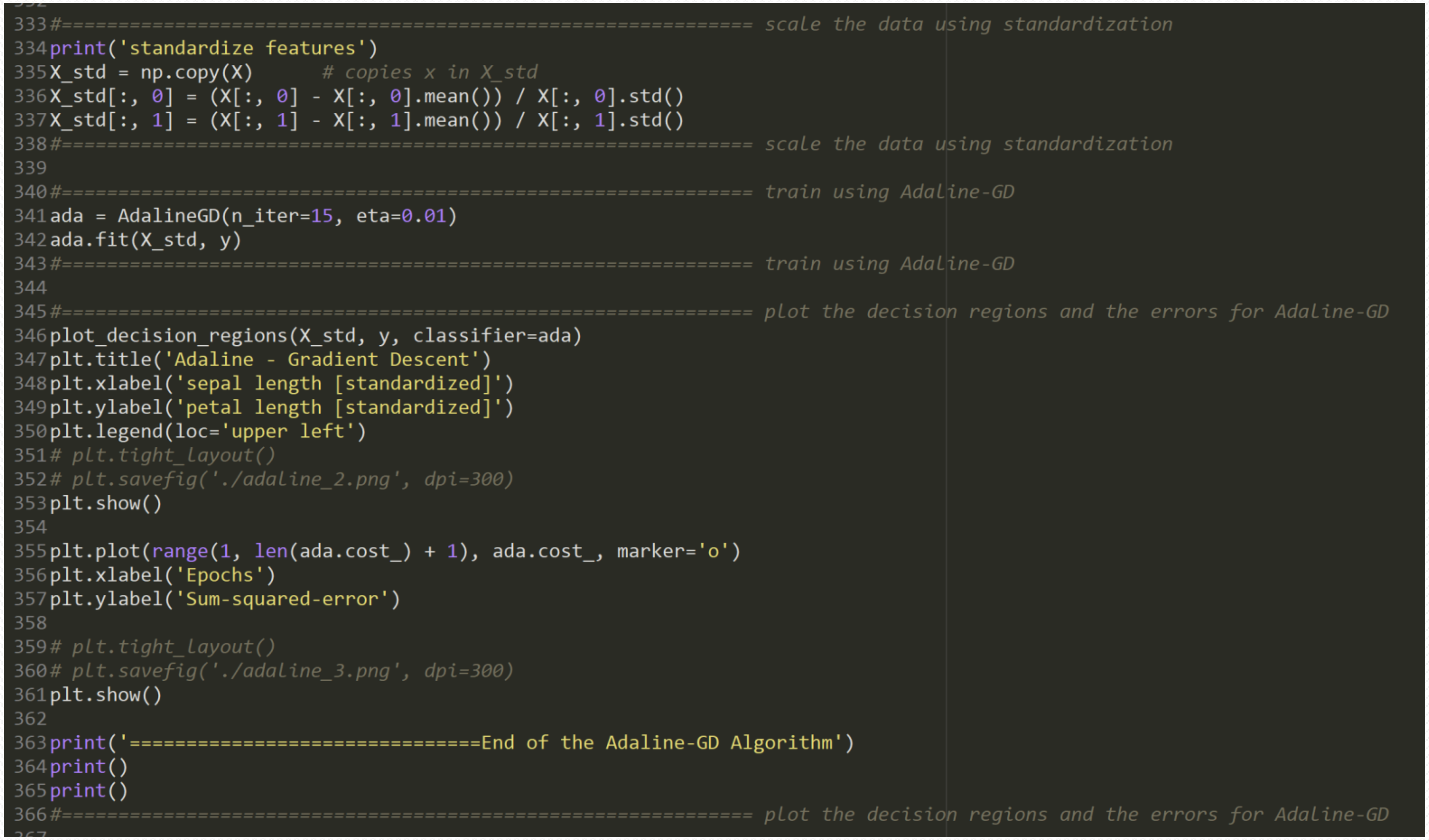
```
        """ Fit training data.

        Parameters
        ----------
        X : {array-like}, shape = [n_samples, n_features]
            Training vectors, where n_samples is the number of samples and
            n_features is the number of features.
        y : array-like, shape = [n_samples]
            Target values.

        Returns
        -------
        self : object

        """
        self.w_ = np.zeros(1 + X.shape[1])          # dimension of w array = number of features + 1
        self.cost_ = []

        for i in range(self.n_iter):
            output = self.net_input(X)
            errors = (y - output)
            self.w_[1:] += self.eta * X.T.dot(errors)
            self.w_[0] += self.eta * errors.sum()
            cost = (errors**2).sum() / 2.0
            self.cost_.append(cost)
        return self
```

Perceptron, Adaline-GD, Adaline-SGD Algorithms



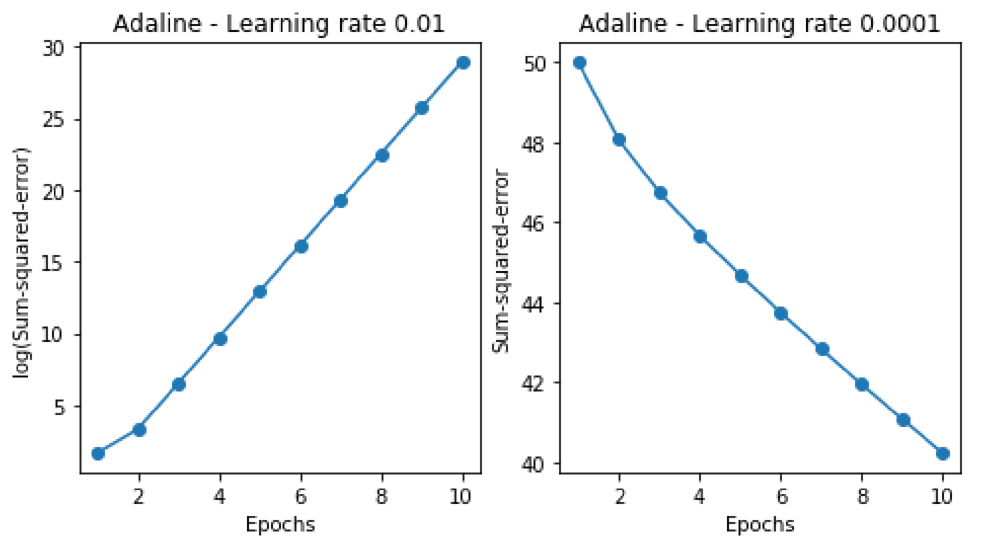
```
    def net_input(self, X):
        """Calculate net input"""
        return np.dot(X, self.w_[1:]) + self.w_[0]

    def activation(self, X):
        """Compute linear activation"""
        return self.net_input(X)

    def predict(self, X):
        """Return class label after unit step"""
        return np.where(self.activation(X) >= 0.0, 1, -1)

#==============================================================  adaline gradient descent (GD) algorithm

fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(8, 4))  # figure with sub-plots (one row, two columns)

#============================================================== call Adaline-GD learning algorithm with large eta
ada1 = AdalineGD(n_iter=10, eta=0.01).fit(X, y)
ax[0].plot(range(1, len(ada1.cost_) + 1), np.log10(ada1.cost_), marker='o')
ax[0].set_xlabel('Epochs')
ax[0].set_ylabel('log(Sum-squared-error)')
ax[0].set_title('Adaline - Learning rate 0.01')
#============================================================== call Adaline-GD learning algorithm with large eta

#============================================================== call Adaline-GD learning algorithm with small eta
ada2 = AdalineGD(n_iter=10, eta=0.0001).fit(X, y)
ax[1].plot(range(1, len(ada2.cost_) + 1), ada2.cost_, marker='o')
ax[1].set_xlabel('Epochs')
ax[1].set_ylabel('Sum-squared-error')
ax[1].set_title('Adaline - Learning rate 0.0001')
#============================================================== call Adaline-GD learning algorithm with small eta

#============================================================== plot the errors for large and small eta(learning rate)
# plt.tight_layout()                                            Adaline-GD
# plt.savefig('./adaline_1.png', dpi=300)
plt.show()                         # show the figure
#============================================================== plot the errors for large and small eta(learning rate)
```

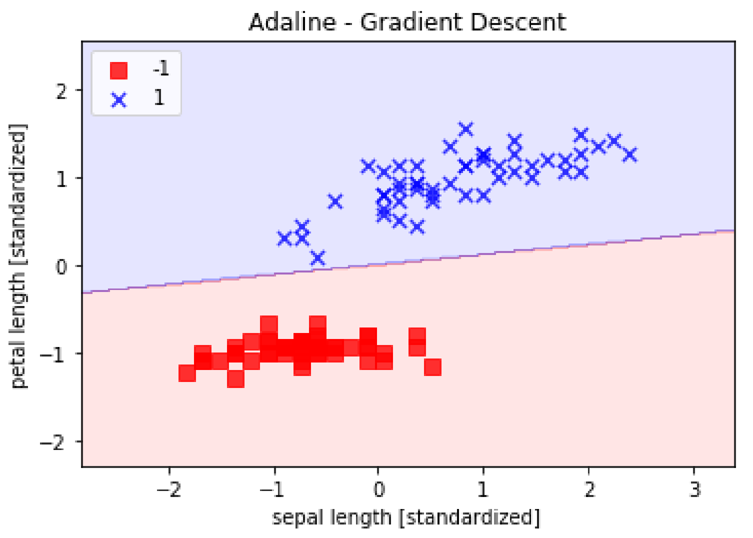
Perceptron, Adaline-GD, Adaline-SGD Algorithms

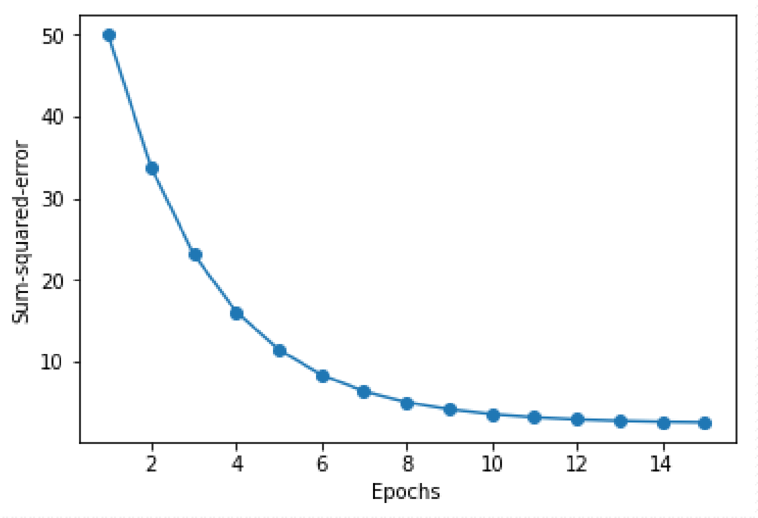


```python
#============================================================= scale the data using standardization
print('standardize features')
X_std = np.copy(X)      # copies x in X_std
X_std[:, 0] = (X[:, 0] - X[:, 0].mean()) / X[:, 0].std()
X_std[:, 1] = (X[:, 1] - X[:, 1].mean()) / X[:, 1].std()
#============================================================= scale the data using standardization

#============================================================= train using Adaline-GD
ada = AdalineGD(n_iter=15, eta=0.01)
ada.fit(X_std, y)
#============================================================= train using Adaline-GD

#============================================================= plot the decision regions and the errors for Adaline-GD
plot_decision_regions(X_std, y, classifier=ada)
plt.title('Adaline - Gradient Descent')
plt.xlabel('sepal length [standardized]')
plt.ylabel('petal length [standardized]')
plt.legend(loc='upper left')
# plt.tight_layout()
# plt.savefig('./adaline_2.png', dpi=300)
plt.show()

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

# plt.tight_layout()
# plt.savefig('./adaline_3.png', dpi=300)
plt.show()

print('===============================End of the Adaline-GD Algorithm')
print()
print()
#============================================================= plot the decision regions and the errors for Adaline-GD
```

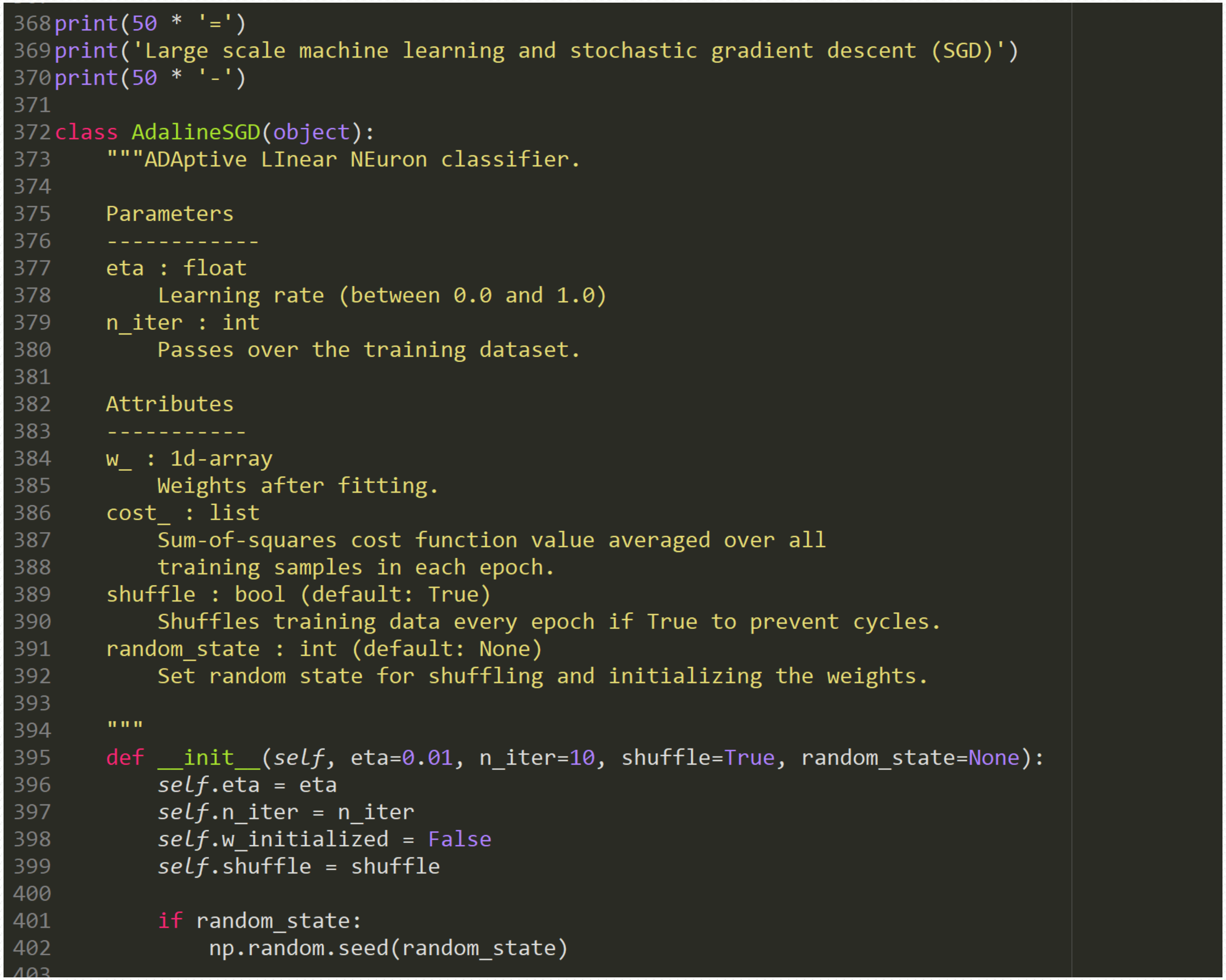
Adaline-GD


Adaline - Learning rate 0.01
log(Sum-squared-error)
30
25
20
15
10
5
2
4
6
8
10
Epochs
Adaline - Learning rate 0.0001
Sum-squared-error
50
48
46
44
42
40
2
4
6
8
10
Epochs


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

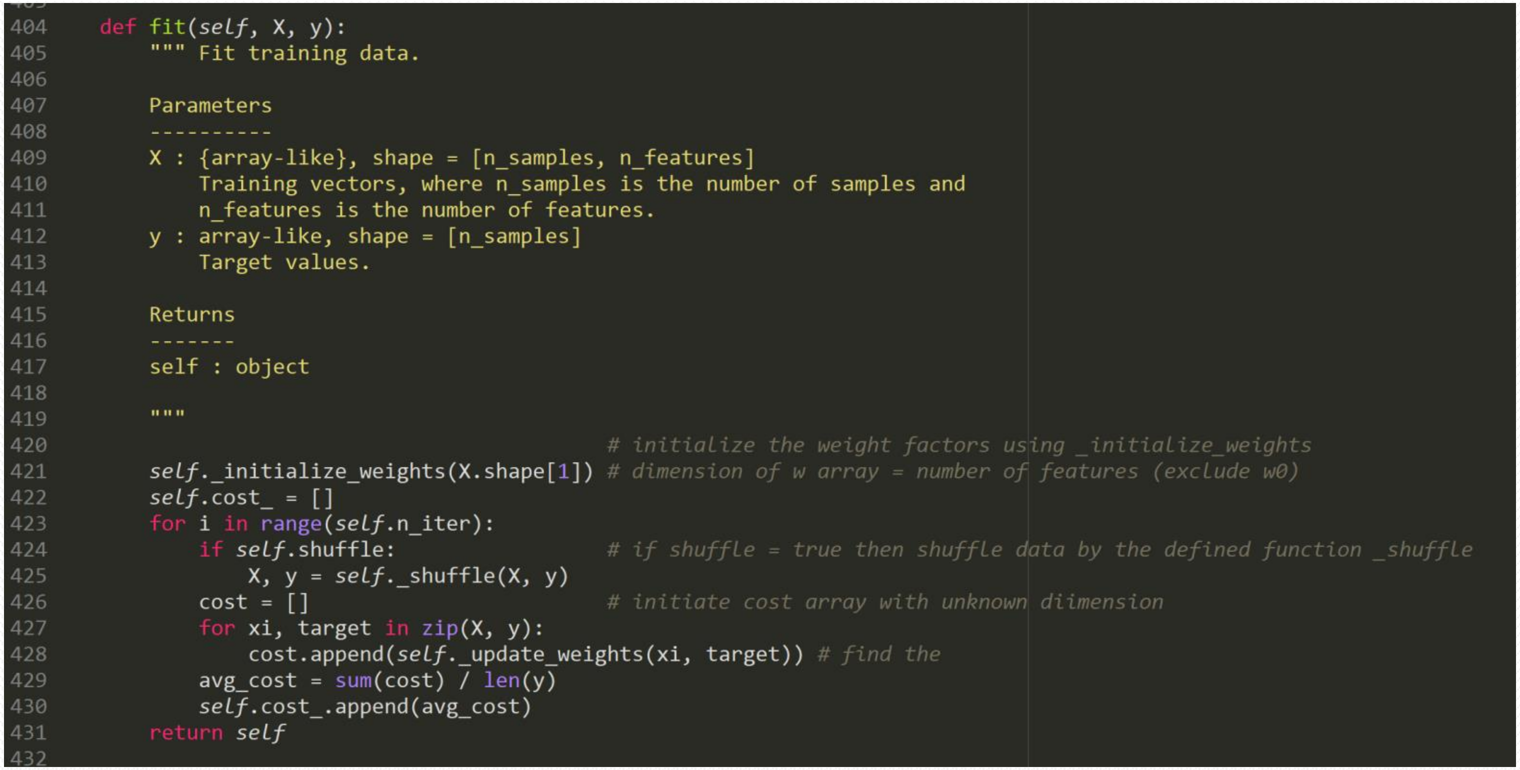

Sum-squared-error
50
40
30
20
10
2
4
6
8
10
12
14
Epochs

Perceptron, Adaline-GD, Adaline-SGD Algorithms



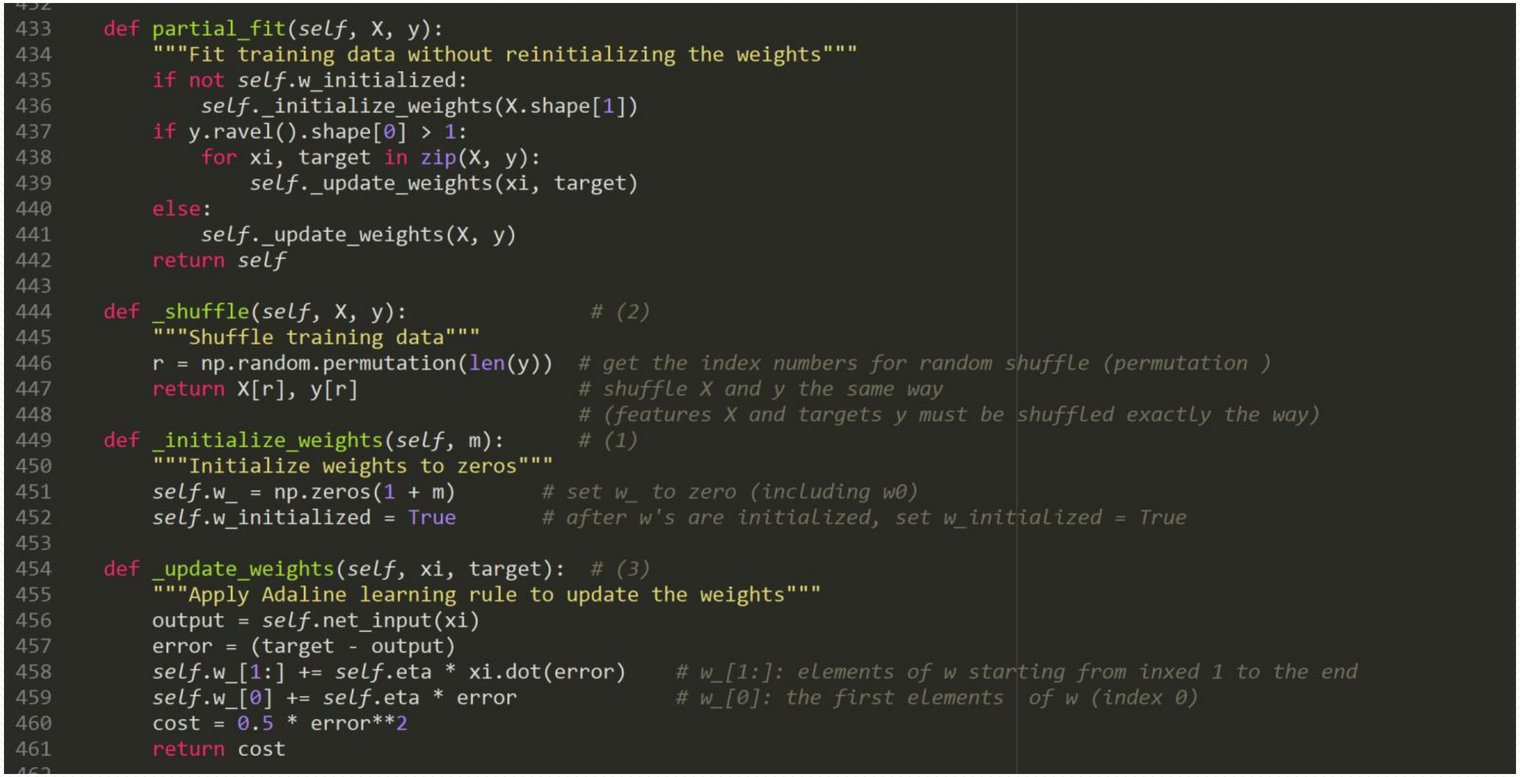
```python
print(50 * '=')
print('Large scale machine learning and stochastic gradient descent (SGD)')
print(50 * '-')

class AdalineSGD(object):
    """ADAptive LInear NEuron classifier.

    Parameters
    ------------
    eta : float
        Learning rate (between 0.0 and 1.0)
    n_iter : int
        Passes over the training dataset.

    Attributes
    -----------
    w_ : 1d-array
        Weights after fitting.
    cost_ : list
        Sum-of-squares cost function value averaged over all
        training samples in each epoch.
    shuffle : bool (default: True)
        Shuffles training data every epoch if True to prevent cycles.
    random_state : int (default: None)
        Set random state for shuffling and initializing the weights.

    """
    def __init__(self, eta=0.01, n_iter=10, shuffle=True, random_state=None):
        self.eta = eta
        self.n_iter = n_iter
        self.w_initialized = False
        self.shuffle = shuffle

        if random_state:
            np.random.seed(random_state)
```

Perceptron, Adaline-GD, Adaline-SGD Algorithms



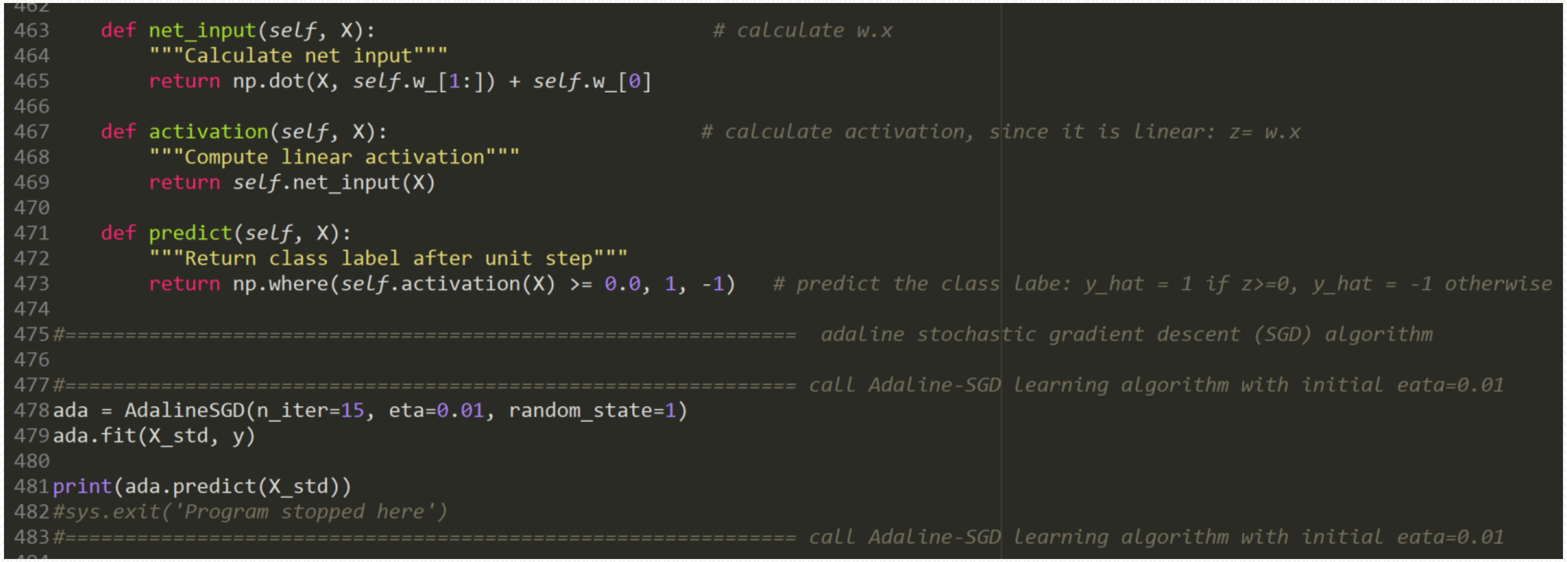
```
    def fit(self, X, y):
        """ Fit training data.

        Parameters
        ----------
        X : {array-like}, shape = [n_samples, n_features]
            Training vectors, where n_samples is the number of samples and
            n_features is the number of features.
        y : array-like, shape = [n_samples]
            Target values.

        Returns
        -------
        self : object

        """
                                             # initialize the weight factors using _initialize_weights
        self._initialize_weights(X.shape[1]) # dimension of w array = number of features (exclude w0)
        self.cost_ = []
        for i in range(self.n_iter):
            if self.shuffle:                 # if shuffle = true then shuffle data by the defined function _shuffle
                X, y = self._shuffle(X, y)
            cost = []                        # initiate cost array with unknown diimension
            for xi, target in zip(X, y):
                cost.append(self._update_weights(xi, target)) # find the
            avg_cost = sum(cost) / len(y)
            self.cost_.append(avg_cost)
        return self

```

Perceptron, Adaline-GD, Adaline-SGD Algorithms



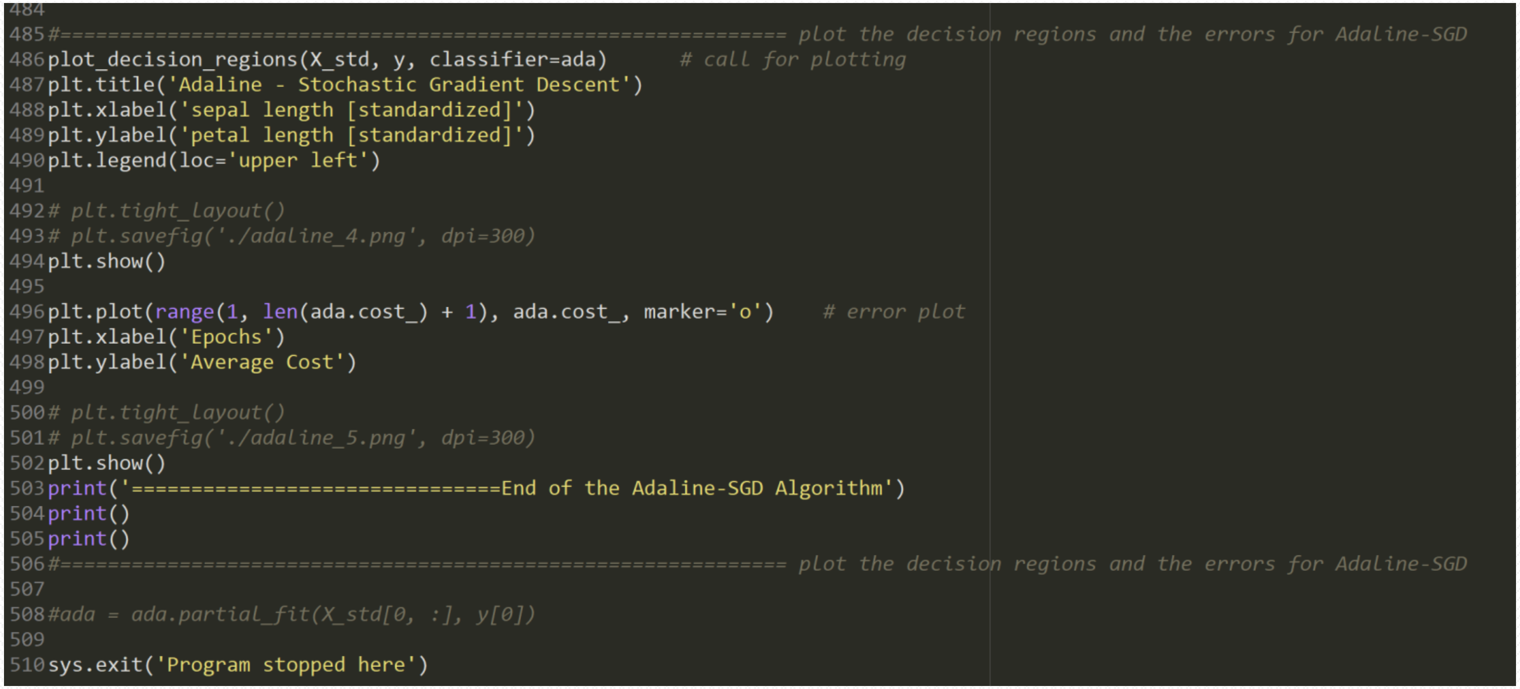
```
    def partial_fit(self, X, y):
        """Fit training data without reinitializing the weights"""
        if not self.w_initialized:
            self._initialize_weights(X.shape[1])
        if y.ravel().shape[0] > 1:
            for xi, target in zip(X, y):
                self._update_weights(xi, target)
        else:
            self._update_weights(X, y)
        return self

    def _shuffle(self, X, y):                   # (2)
        """Shuffle training data"""
        r = np.random.permutation(len(y))  # get the index numbers for random shuffle (permutation )
        return X[r], y[r]                  # shuffle X and y the same way
                                           # (features X and targets y must be shuffled exactly the way)
    def _initialize_weights(self, m):      # (1)
        """Initialize weights to zeros"""
        self.w_ = np.zeros(1 + m)      # set w_ to zero (including w0)
        self.w_initialized = True      # after w's are initialized, set w_initialized = True

    def _update_weights(self, xi, target):  # (3)
        """Apply Adaline learning rule to update the weights"""
        output = self.net_input(xi)
        error = (target - output)
        self.w_[1:] += self.eta * xi.dot(error)    # w_[1:]: elements of w starting from inxed 1 to the end
        self.w_[0] += self.eta * error             # w_[0]: the first elements  of w (index 0)
        cost = 0.5 * error**2
        return cost
```

Perceptron, Adaline-GD, Adaline-SGD Algorithms



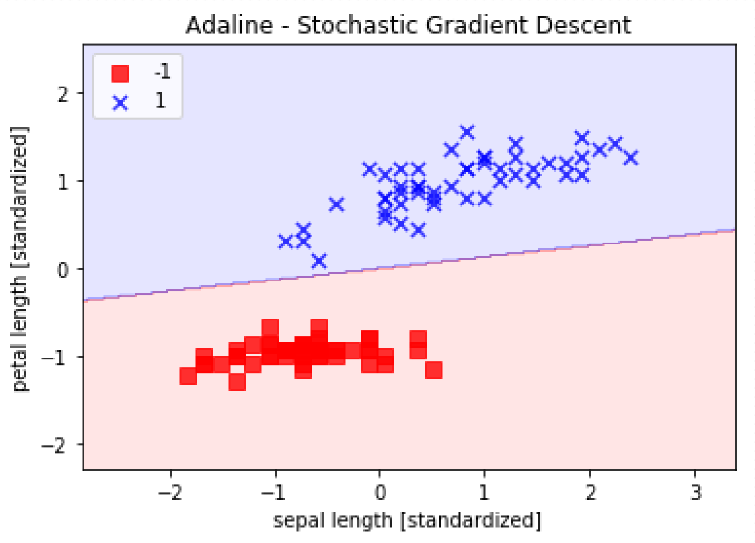
```
    def net_input(self, X):                            # calculate w.x
        """Calculate net input"""
        return np.dot(X, self.w_[1:]) + self.w_[0]

    def activation(self, X):                          # calculate activation, since it is linear: z= w.x
        """Compute linear activation"""
        return self.net_input(X)

    def predict(self, X):
        """Return class label after unit step"""
        return np.where(self.activation(X) >= 0.0, 1, -1)   # predict the class labe: y_hat = 1 if z>=0, y_hat = -1 otherwise

#=============================================================  adaline stochastic gradient descent (SGD) algorithm

#============================================================= call Adaline-SGD learning algorithm with initial eata=0.01
ada = AdalineSGD(n_iter=15, eta=0.01, random_state=1)
ada.fit(X_std, y)

print(ada.predict(X_std))
#sys.exit('Program stopped here')
#============================================================= call Adaline-SGD learning algorithm with initial eata=0.01
```

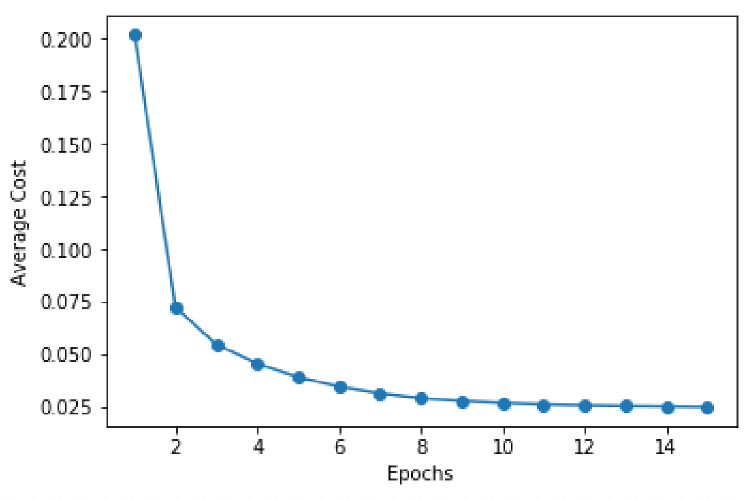
Perceptron, Adaline-GD, Adaline-SGD Algorithms



```python

#============================================================== plot the decision regions and the errors for Adaline-SGD
plot_decision_regions(X_std, y, classifier=ada)      # call for plotting
plt.title('Adaline - Stochastic Gradient Descent')
plt.xlabel('sepal length [standardized]')
plt.ylabel('petal length [standardized]')
plt.legend(loc='upper left')

# plt.tight_layout()
# plt.savefig('./adaline_4.png', dpi=300)
plt.show()

plt.plot(range(1, len(ada.cost_) + 1), ada.cost_, marker='o')     # error plot
plt.xlabel('Epochs')
plt.ylabel('Average Cost')

# plt.tight_layout()
# plt.savefig('./adaline_5.png', dpi=300)
plt.show()
print('===============================End of the Adaline-SGD Algorithm')
print()
print()
#============================================================== plot the decision regions and the errors for Adaline-SGD

#ada = ada.partial_fit(X_std[0, :], y[0])

sys.exit('Program stopped here')
```

Adaline-SGD


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


Average Cost
0.200
0.175
0.150
0.125
0.100
0.075
0.050
0.025
2
4
6
8
10
12
14
Epochs