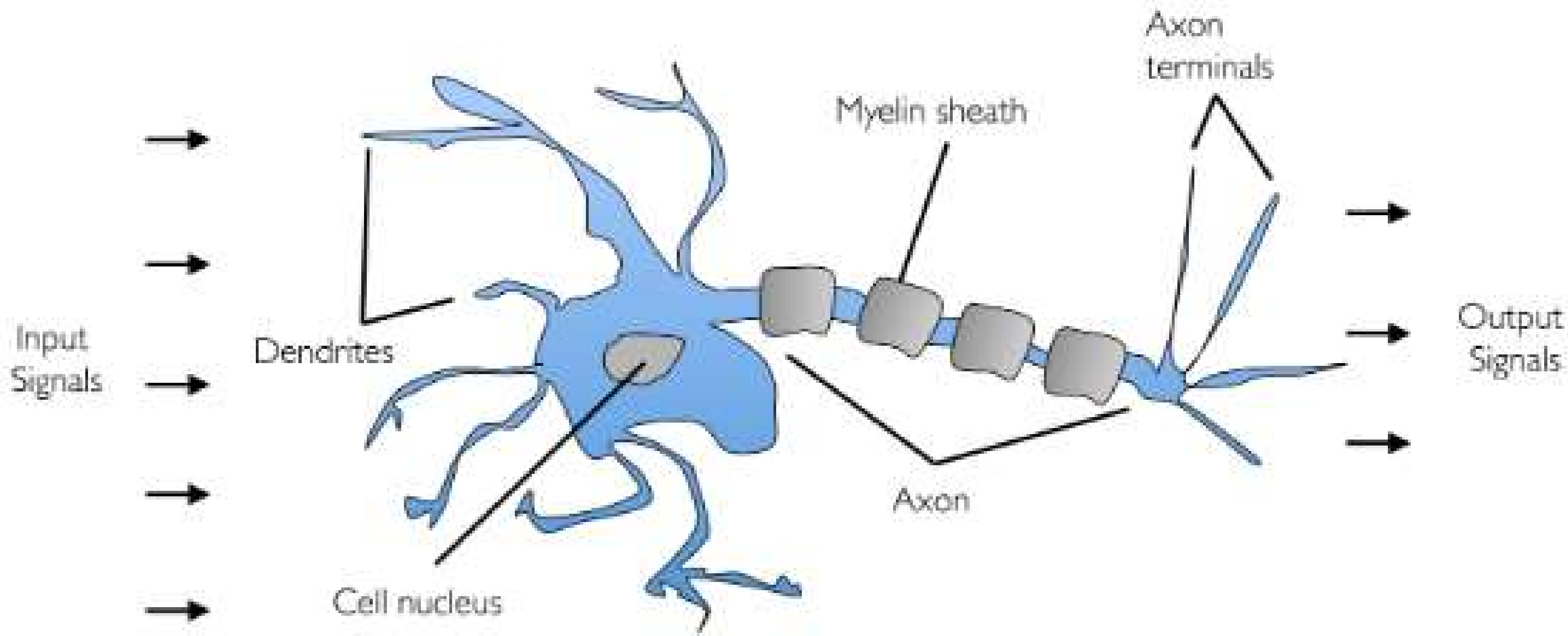
Gradient Descent

Machine Learning

The Perceptron

- Artificial Neurons



The Perceptron

- The Model
  - Input, weights:

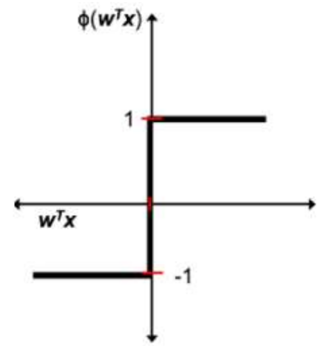
$$\mathbf{x} = \begin{bmatrix} x_1 \\ \cdot \\ \cdot \\ x_m \end{bmatrix} \quad \mathbf{w} = \begin{bmatrix} w_1 \\ \cdot \\ \cdot \\ w_m \end{bmatrix}$$

Let $x_0 = 1,\qquad w_0 = \text{-T}$

  - Output function:

$$z = w_0x_0 + w_1x_1 + \cdots + w_mx_m = \mathbf{w}^{\mathrm{T}}\mathbf{x}$$

$$\hat{y} = \phi(z) = \begin{cases} 1 & if\ \ z \geq 0 \\ -1 & otherwise \end{cases}$$



dongguk UNIVERSITY

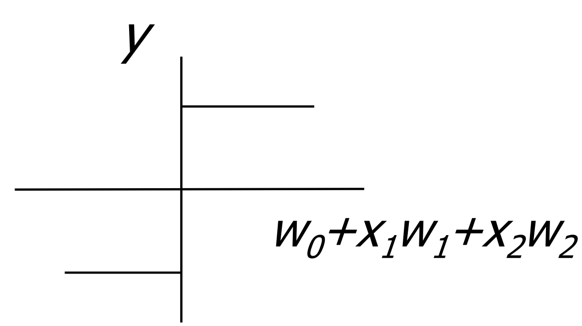
Perceptron Learning

- Learning
  - Training data: Set of <**x**, *y*>
  1. Initialize random **w**, learning rate η
  2. For each **x**,
     1. compute $\hat{y}$ $(1 \text{ if } \mathbf{w}^{\mathrm{T}}\mathbf{x} \geq 0)$
     2. Update **w**

$$w_j = w_j + \Delta w_j$$

$$\boxed{\Delta w_j = \eta(y - \hat{y})x_j}$$

  - If y = 1 but $\hat{y}$ = -1 → increase weights of + input
  - If y = -1 but $\hat{y}$ = 1 → decrease weights of + input



dongguk UNIVERSITY

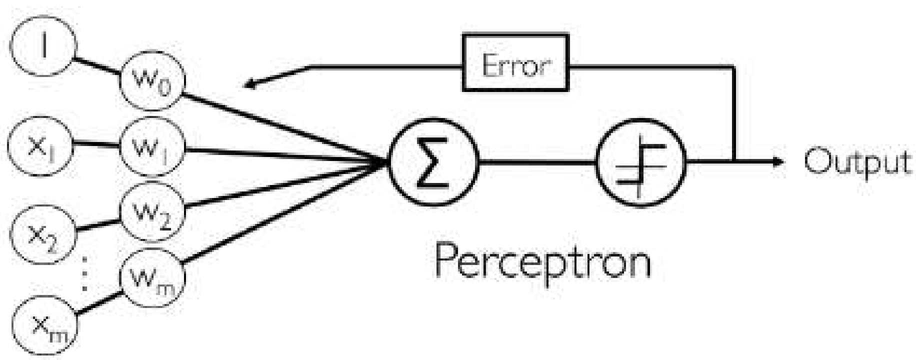
Perceptron Learning

- Example ($\eta\ = 0.1$)
  - If training data 1 ($x_1 = 0.4, x_2 = -0.5,\ y = -1$) → $\hat{y} = -1$

$$\Delta w_j = \eta(y - \hat{y})x_j = 0.1(-1 - (-1))x_j = 0 \quad \text{(No change)}$$

  - If training data 2 ($x_1 = 0.5, x_2 = -0.4,\ y = -1$) → $\hat{y} = 1$

$$\Delta w_0 = \eta(y - \hat{y})x_0 = 0.1(-1 - (1))1 = -0.2$$
$$\Delta w_1 = \eta(y - \hat{y})x_1 = 0.1(-1 - (1))0.5 = -0.1$$
$$\Delta w_2 = \eta(y - \hat{y})x_2 = 0.1(-1 - (1))\ (-0.4) = +0.08$$



Perceptron

- Load Iris dataset

```
import pandas as pd

df = pd.read_csv('iris.csv', header=None)
df.tail()
```

Iris dataset

0 : sepal length(cm)
1 : sepal width(cm)
2 : petal length(cm)
3 : petal length(cm)
4 : class
  - Iris Setosa
  - Iris Versicolour
  - Iris Virginica

Out[2]:

	0	1	2	3	4
145	6.7	3.0	5.2	2.3	Iris-virginica
146	6.3	2.5	5.0	1.9	Iris-virginica
147	6.5	3.0	5.2	2.0	Iris-virginica
148	6.2	3.4	5.4	2.3	Iris-virginica
149	5.9	3.0	5.1	1.8	Iris-virginica

Perceptron

- Preprocessing for training data

```
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np

# select setosa and versicolor
y = df.iloc[0:100, 4].values
y
```

```
Out[5]: array(['Iris-setosa', 'Iris-setosa', 'Iris-setosa', 'Iris-setosa',
               'Iris-setosa', 'Iris-setosa', 'Iris-setosa', 'Iris-setosa',
               'Iris-setosa', 'Iris-setosa', 'Iris-setosa', 'Iris-setosa',
               'Iris-setosa', 'Iris-setosa', 'Iris-setosa', 'Iris-setosa',
               'Iris-setosa', 'Iris-setosa', 'Iris-setosa', 'Iris-setosa',
               'Iris-setosa', 'Iris-setosa', 'Iris-setosa', 'Iris-setosa',
               'Iris-setosa', 'Iris-setosa', 'Iris-setosa', 'Iris-setosa',
               'Iris-setosa', 'Iris-setosa', 'Iris-setosa', 'Iris-setosa',
               'Iris-setosa', 'Iris-setosa', 'Iris-setosa', 'Iris-setosa',
```

```
# Change the values (setosa = -1, virginica=1)
y = np.where(y == 'Iris-setosa', -1, 1)
y
```

```
Out[6]: array([-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
               -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
               -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,  1,
                1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
                1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
                1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1])
```

Perceptron

- Preprocessing for training data

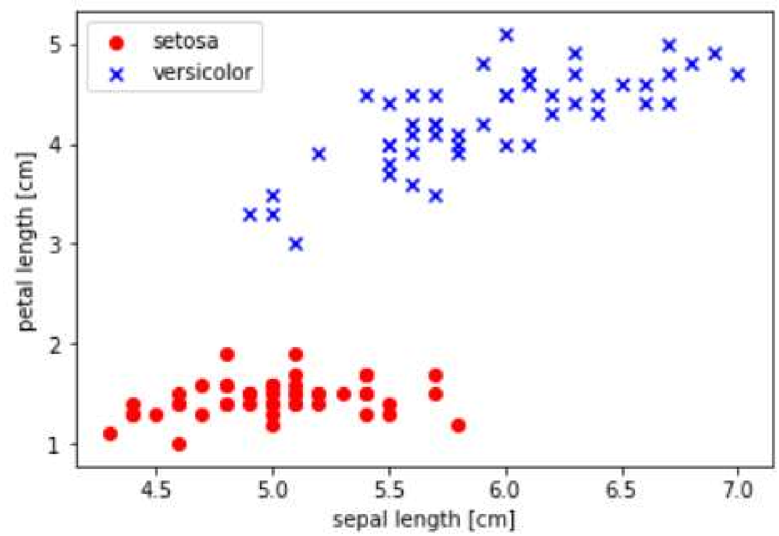
```python
# extract sepal length and petal length
X = df.iloc[0:100, [0, 2]].values
X
```

```
Out[7]: array([[5.1, 1.4],
               [4.9, 1.4],
               [4.7, 1.3],
               [4.6, 1.5],
               [5. , 1.4],
               [5.4, 1.7]
```

- Plotting the data

```python
# 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.show()
```



Perceptron

- Define Perceptron Class

```python
class Perceptron(object):
    def __init__(self, eta=0.01, n_iter=50, random_state=1):
        self.eta = eta # learning rate
        self.n_iter = n_iter # number of iteration
        self.random_state = random_state # random generator seed for random weight

        # weight initiailization
        rgen = np.random.RandomState(self.random_state)
        self.w_ = rgen.normal(loc=0.0, scale=0.01, size=1 + X.shape[1])

    def fit(self, X, y):
        self.errors_ = []
        for _ in range(self.n_iter):
            errors = 0
            for xi, target in zip(X, y):
                # wj =  wj + eta * (y - yhat) * xj
                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):
        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)
```

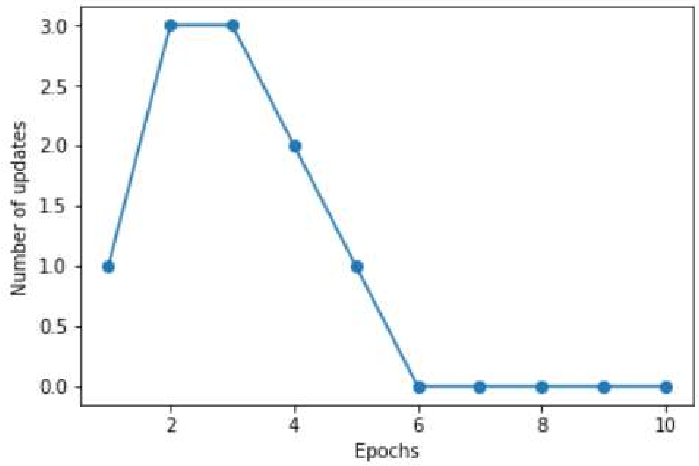
Perceptron

- Training the perceptron model
  - Training a perceptron model using given Perceptron Class

```
# Training Perceptron
model = Perceptron(eta=0.1, n_iter=10)

model.fit(X, y)

# Plotting the number of errors
plt.plot(range(1, len(model.errors_) + 1), model.errors_, marker='o')
plt.xlabel('Epochs')
plt.ylabel('Number of updates')
plt.show()
```

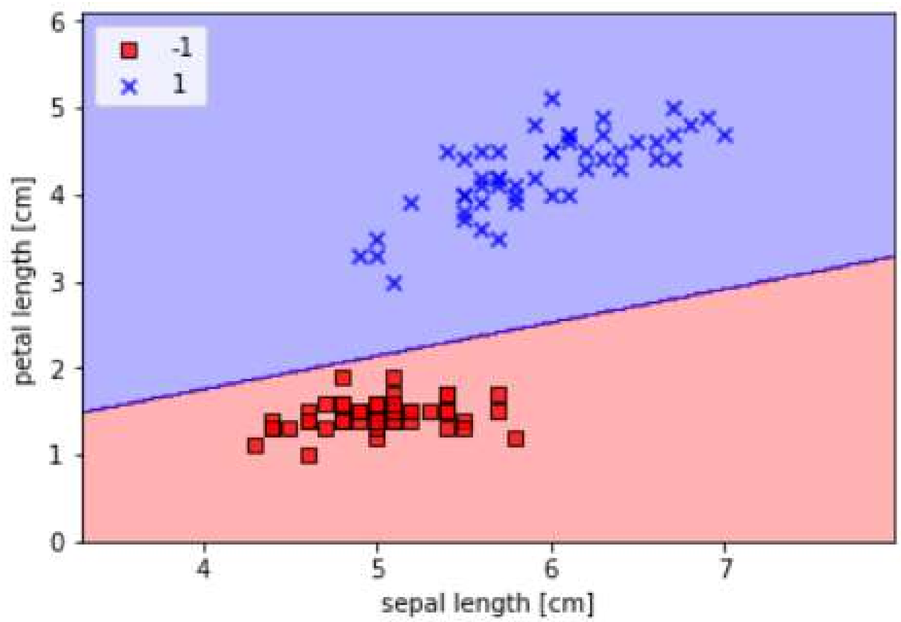


Perceptron

- Training a perceptron model
  - Visualizing using given decision regions function

```
plot_decision_regions(X, y, classifier=model)
plt.xlabel('sepal length [cm]')
plt.ylabel('petal length [cm]')
plt.legend(loc='upper left')

plt.show()
```



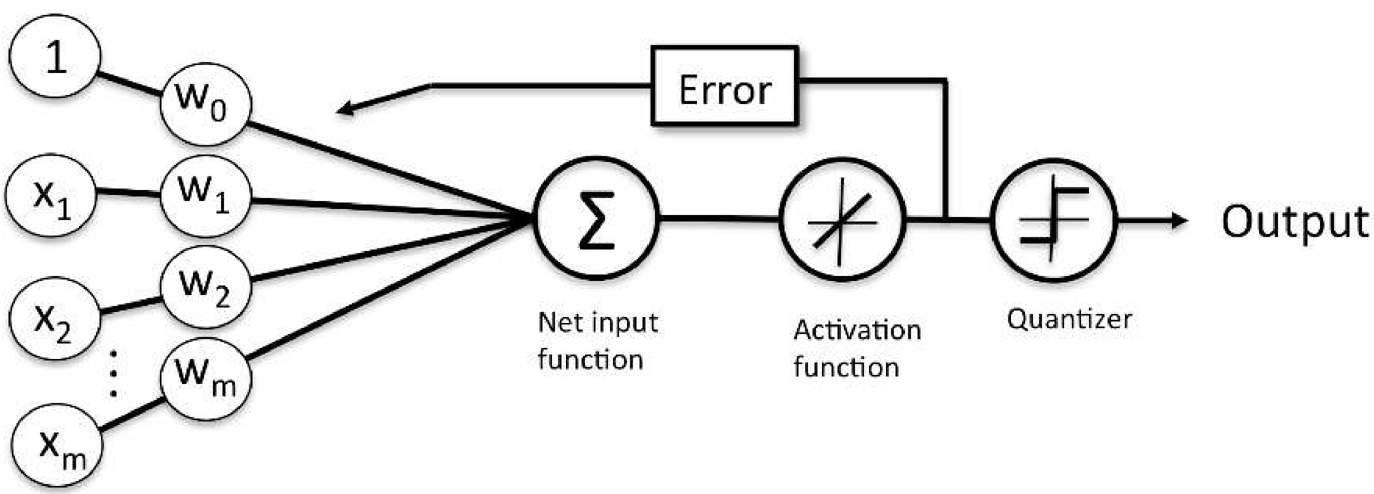
Adaline : Adaptive Linear Neurons

- Adaptive linear neuron model
  - Use linear output function :

$$z = w_0 x_0 + w_1 x_1 + \cdots + w_m x_m = \mathbf{w}^{\mathrm{T}}\mathbf{x}$$

$$\hat{y} = 1 \;\; if \;\; \phi(z) = z \geq 0$$

  - Define ***cost function*** and minimize it



Cost Function

- Cost function / loss function
  - A measure of how wrong the model is in terms of its ability to estimate the relationship between X and y
  - Goal of learning algorithm is to ***minimize the cost function***
- Cost function of Adaline
  - Represent the error

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

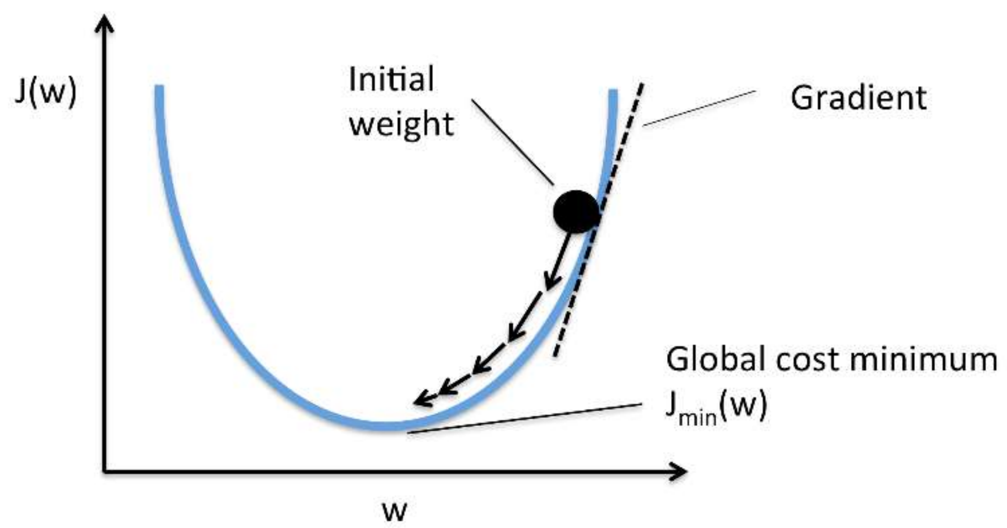
➡ How to minimize it?

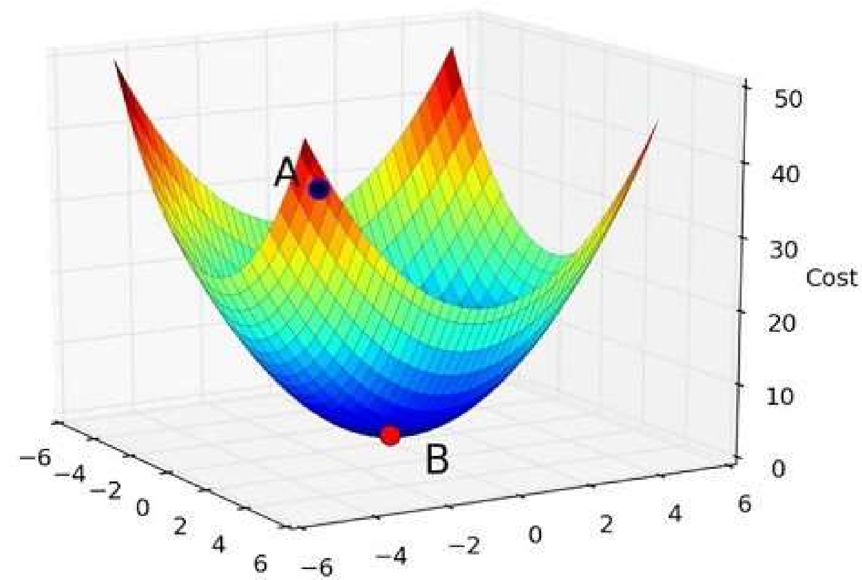
(How to change weights w?)

Gradient Descent

- Gradient descent (steepest descent)
  - An iterative optimization algorithm for finding the minimum of a function
  - One takes steps proportional to the ***negative of the gradient***

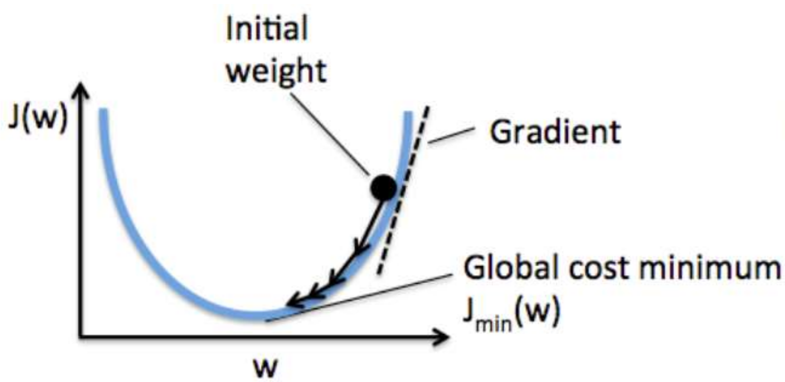
$$\mathbf{w}_{n+1} = \mathbf{w}_n - \eta \nabla J(\mathbf{w}) \qquad \nabla J(\mathbf{w}) = \left(\frac{\partial J}{\partial w_0}, \frac{\partial J}{\partial w_1}, \dots, \frac{\partial J}{\partial w_m}\right)$$

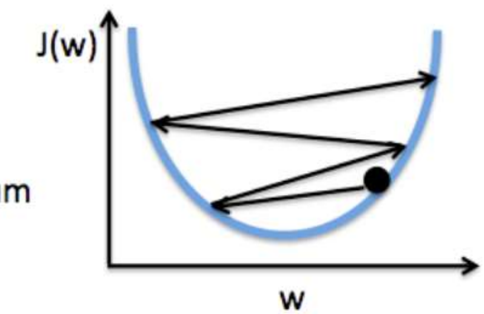


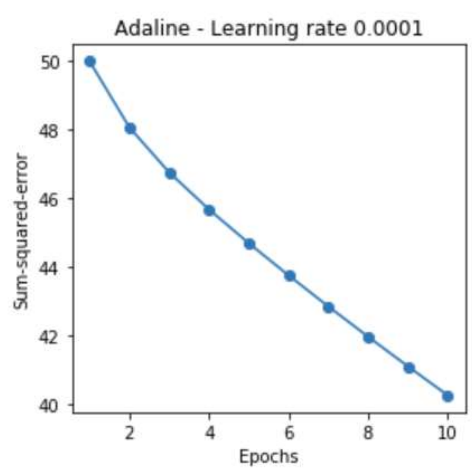


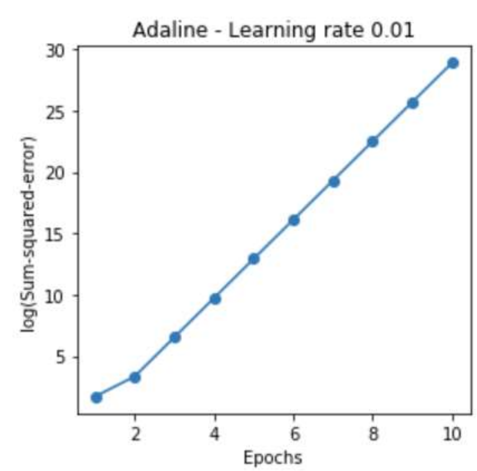
Gradient Descent

- Effect of learning rate η









Adaline : Adaptive Linear Neurons

- Define Adaline Class – (1)

```
class AdalineGD(object):
        def __init__(self, eta=0.01, n_iter=50, random_state=1):
                self.eta = eta # learning rate
                self.n_iter = n_iter # number of iteration
                self.random_state = random_state

                # weight initiailization
                rgen = np.random.RandomState(self.random_state)
                self.w_ = rgen.normal(loc=0.0, scale=0.01, size=1 + X.shape[1])

        def fit(self, X, y):
                self.cost_ = []

                for i in range(self.n_iter):
                        net_input = self.net_input(X)
                        output = self.activation(net_input)

                        # w = w + eta * (X.T dot errors)
                        errors = (y - output)
                        self.w_[1:] += self.eta * X.T.dot(errors)
                        self.w_[0] += self.eta * errors.sum()

                        # compute cost
                        cost = (errors**2).sum() / 2.0
                        self.cost_.append(cost)
                        print(self.w_)

                return self
```

Adaline : Adaptive Linear Neurons

- Define Adaline Class – (2)

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

    def activation(self, X):
        return X

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

Adaline : Adaptive Linear Neurons

- Standardize features

```
# 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()
X_std
```

```
Out[179]:  array([[-0.5810659 , -1.01435952],
                  [-0.89430898, -1.01435952],
                  [-1.20755205, -1.08374115],
                  [-1.36417359, -0.94497788],
                  [-0.73768744, -1.01435952],
                  [-0.11120129, -0.80621461],
                  [-1.36417359, -1.01435952],
                  [-0.73768744, -0.94497788],
                  [-1.67741667, -1.01435952],
                  [-0.89430898, -0.94497788],
                  [-0.11120129, -0.94497788],
                  [-1.05093052, -0.87559625],
                  [-1.05093052, -1.01435952],
                  [-1.8340382 , -1.22250442],
                  [ 0.51528486, -1.15312279],
```

⋮

Adaline : Adaptive Linear Neurons

- Training an Adaline model on the Iris dataset
  - Training AdalineGD with learning rate 0.1, 0.0001, and 0.01
  - Plotting the cost graph
  - Visualizing the model using given decision regions function
  - Computing the accuracy of the model

```
# Training AdalineGD with learning rate 0.1
ada1 = AdalineGD(n_iter=10, eta=0.1)
ada1.fit(X_std, y)
```

```
[-0.14619108   7.38086767   9.79678659]
[    1.31571974 -138.73294518 -138.43289951]
[  -11.8414777    2380.49335345   2382.64713229]
[    106.57329931 -40773.52560914 -40772.99590048]
⋮
```

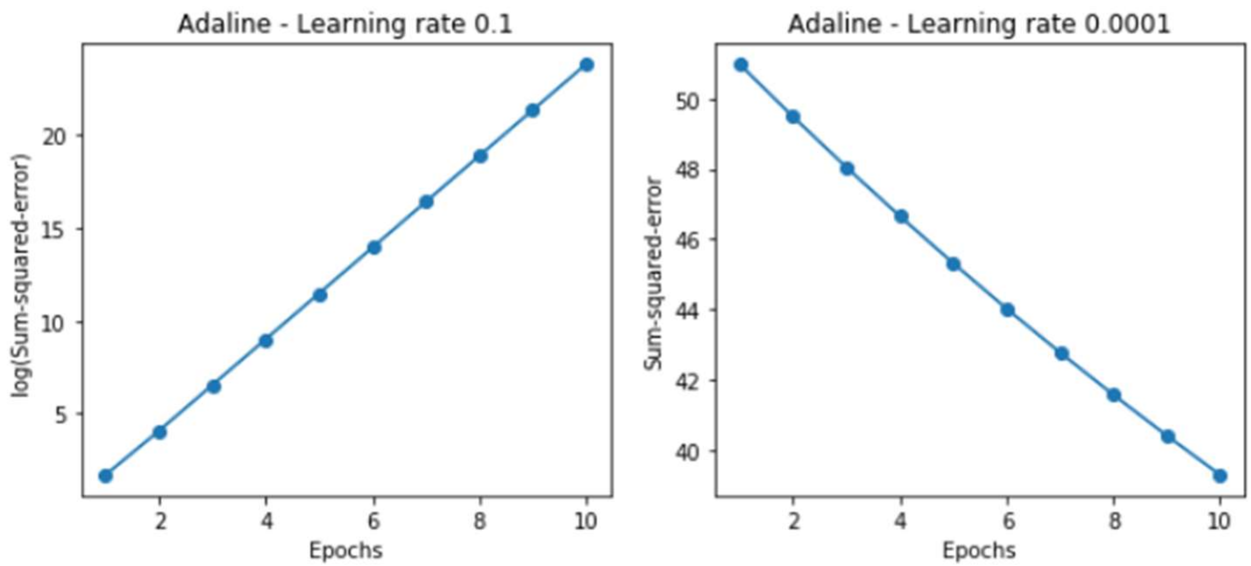
```
# Training AdalineGD with learning rate 0.0001
ada2 = AdalineGD(n_iter=10, eta=0.0001)
ada2.fit(X_std, y)
```

```
[ 0.01608102  0.00126942  0.00452035]
[ 0.01592021  0.00850291  0.01416439]
[ 0.01576101  0.01558571  0.02365322]
[ 0.0156034   0.0225206   0.03298962]
⋮
```

Adaline : Adaptive Linear Neurons

- Training an Adaline model on the Iris dataset

```
# Plotting cost
fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(10, 4))
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 0.1')
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')
plt.show()
```



dongguk UNIVERSITY

Adaline : Adaptive Linear Neurons

- Training an Adaline model on the Iris dataset

```
y
```

```
Out[34]: array([-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1
               -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1
               -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,  1
                1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1
                1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1
                1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1])
```

```
ada1.predict(X_std)
```

```
Out[32]: array([ 1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
                1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
                1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1, -1,
               -1, -1, -1, -1, -1, -1,  1, -1, -1,  1, -1, -1, -1, -1, -1, -1, -1,
               -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
               -1, -1, -1, -1, -1, -1, -1, -1,  1, -1, -1, -1, -1,  1, -1])
```

```
ada2.predict(X_std)
```

```
Out[33]: array([-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
               -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
               -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,  1,
                1,  1,  1,  1,  1,  1, -1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
                1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
                1,  1,  1,  1,  1,  1,  1,  1, -1,  1,  1,  1,  1, -1,  1])
```

Adaline : Adaptive Linear Neurons

- Training an Adaline model on the Iris dataset

```
# Training AdalineGD with learning rate 0.01
ada = AdalineGD(n_iter=20, eta=0.01)
ada.fit(X_std, y)
```

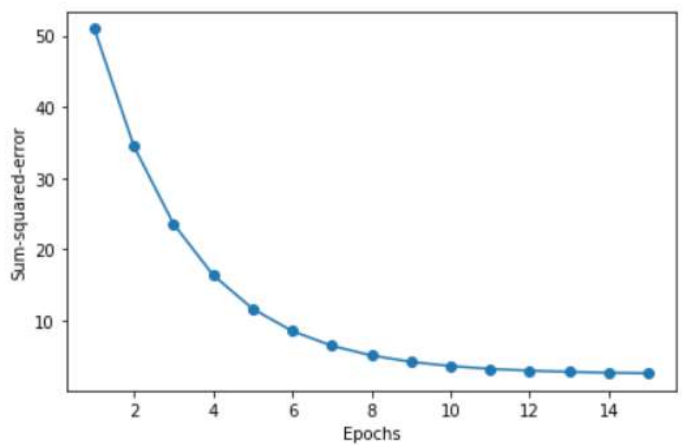
```
[ 0.          0.73258096  0.97492511]
[  1.47437618e-15  -6.37284985e-02   3.74814400e-01]
[  1.66533454e-17   4.23794973e-01   1.02172761e+00]
[  1.06248343e-15  -1.01750346e-01   6.25668805e-01]
[  5.99520433e-17   2.20003559e-01   1.05261615e+00]
[  7.69384556e-16  -1.26843862e-01   7.91226751e-01]
[  9.43689571e-17   8.55060696e-02   1.07300186e+00]
[  6.08402217e-16  -1.43404986e-01   9.00491061e-01]
[  1.48769885e-16  -3.25907753e-03   1.08645593e+00]
[  3.67483821e-16  -1.54334935e-01   9.72602910e-01]
```

⋮

Adaline : Adaptive Linear Neurons

- Training an Adaline model on the Iris dataset

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



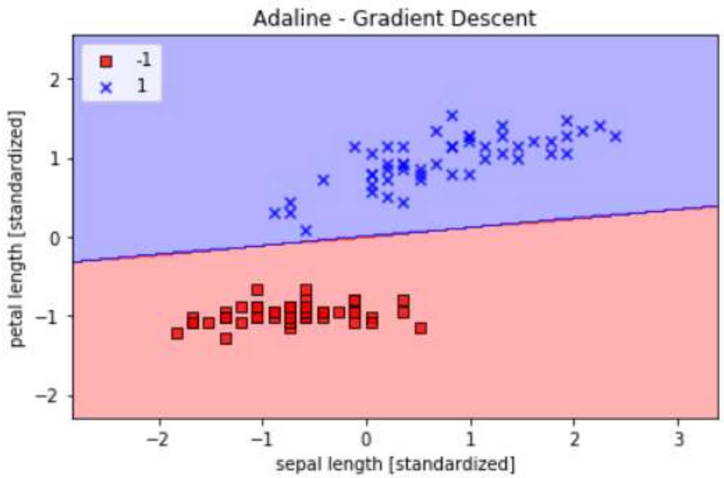

```
ada.predict(X_std)
```

```
Out[37]: array([-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
                -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
                -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,  1,
                 1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
                 1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
                 1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1])
```

Adaline : Adaptive Linear Neurons

- Training an Adaline model on the Iris dataset

```
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.show()
```




```
# Computing the accuracy of the model
y_pred = ada.predict(X_std)
accuracy = np.sum(y == y_pred)/len(y)
print("Accuracy on the training set =", accuracy)
```

```
Accuracy on the training set = 1.0
```

Adaline with Stochastic Gradient Descent

- GD
  - Compute gradient with entire training dataset **X** → update **w**
  - When training dataset is large (ex> 1,000,000), 1 weight update needs large amount of computation
- SGD
  - Compute gradient with 1 training data **xi** → update **w**
  - Approximation of gradient causes noise
- Mini-batch SGD
  - Compute gradient with a small subset of training dataset **b** → update **w** (ex> b = 32)
  - Less noise
  - More efficient computation

Adaline with Stochastic Gradient Descent

- Define AdalineSGD Class – (1)

```
class AdalineSGD(object):
    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
        self.random_state = random_state
        self._initialize_weights(X.shape[1])

    def fit(self, X, y):
        self.cost_ = []
        for i in range(self.n_iter):
            if self.shuffle:
                X, y = self._shuffle(X, y)
            cost = []
            for xi, target in zip(X, y):
                cost.append(self._update_weights(xi, target))
            avg_cost = sum(cost) / len(y)
            self.cost_.append(avg_cost)
        return self
```

⋮

Adaline with Stochastic Gradient Descent

- Define AdalineSGD Class – (2)

```
def _shuffle(self, X, y):
    r = self.rgen.permutation(len(y))
    return X[r], y[r]

def _initialize_weights(self, m):
    self.rgen = np.random.RandomState(self.random_state)
    self.w_ = self.rgen.normal(loc=0.0, scale=0.01, size=1 + m)
    self.w_initialized = True

def _update_weights(self, xi, target):
    output = self.activation(self.net_input(xi))
    error = (target - output)
    self.w_[1:] += self.eta * xi.dot(error)
    self.w_[0] += self.eta * error
    cost = 0.5 * error**2
    return cost

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

def activation(self, X):
    return X

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

Adaline with Stochastic Gradient Descent

- Training an Adaline model with SGD on the iris dataset
  - Training and visualizing a Adaline model using AdalineSGD Class

```
ada = AdalineSGD(n_iter=15, eta=0.01, random_state=1)
ada.fit(X_std, y)
```

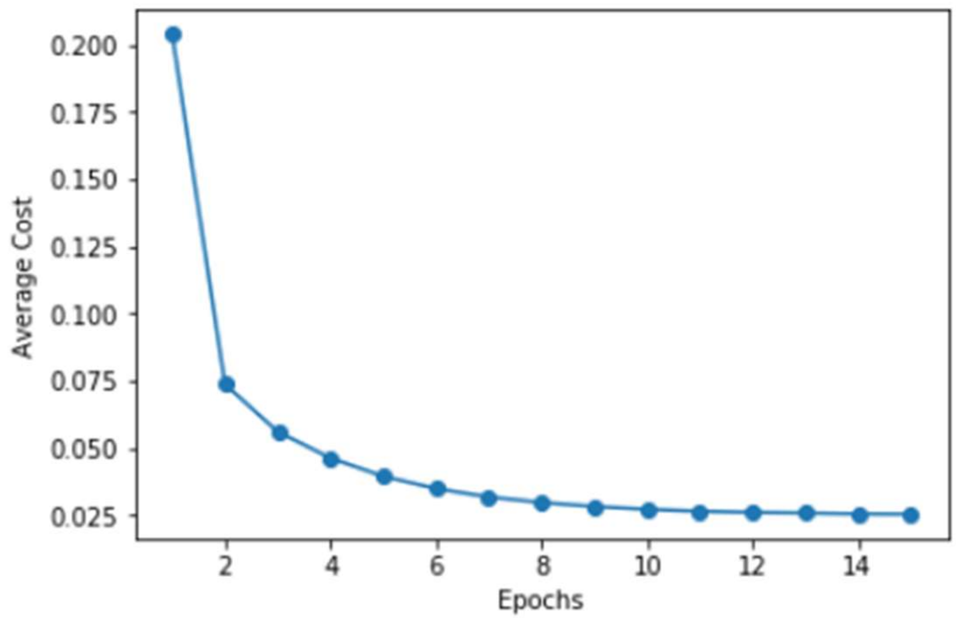
```
[ 0.00547304  0.27285356  0.49825661]
[-0.00608069  0.23654068  0.64462437]
[ 0.00926127  0.18523628  0.74504939]
[ 0.0015176   0.12647723  0.81131789]
[ 0.00386798  0.07201719  0.86046615]
[ 0.00740254  0.0312768   0.90627882]
[  7.88472441e-03   4.13494918e-04   9.46872499e-01]
[ 0.00251789 -0.03650693  0.96815509]
[-0.00110447 -0.06194687  0.99205647]
[-0.00581369 -0.08027863  1.01350822]
[-0.00954177 -0.09770813  1.02772661]
[-0.00632273 -0.11472444  1.03770746]
[-0.00587553 -0.12175986  1.052657  ]
[-0.00632225 -0.13560041  1.05923075]
[  2.27202773e-04  -1.38544756e-01   1.07263215e+00]

<__main__.AdalineSGD at 0x26b8e82d0f0>
```

Adaline with Stochastic Gradient Descent

- Training an Adaline model with SGD on the iris dataset
  - Training and visualizing a Adaline model using AdalineSGD Class

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



Adaline with Stochastic Gradient Descent

- Training an Adaline model with SGD on the iris dataset
  - Training and visualizing a Adaline model using AdalineSGD Class

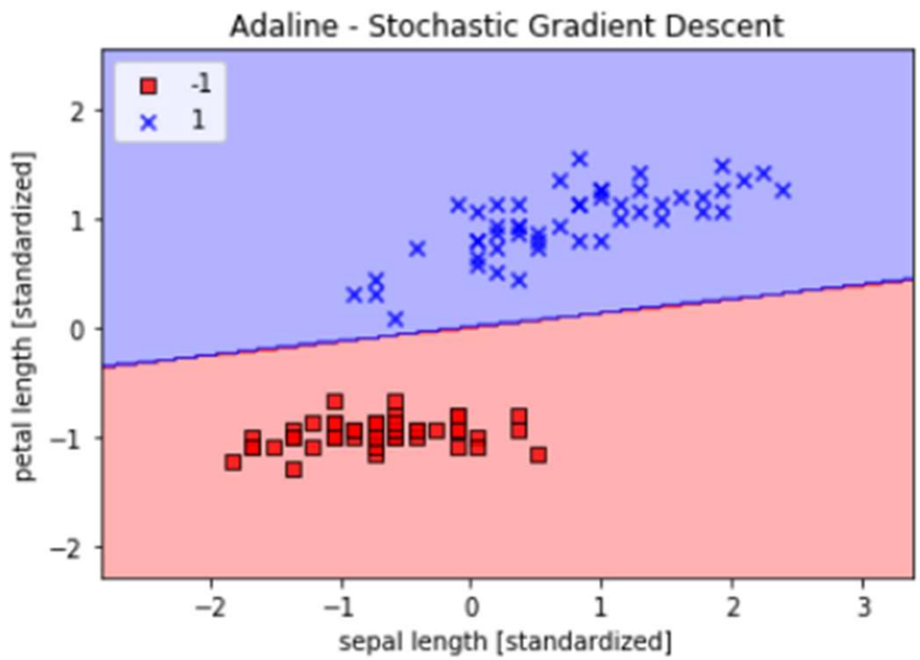
```
ada.predict(X_std)
```

```
Out[195]: array([-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
                 -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,
                 -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1,  1,
                  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
                  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
                  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1])
```

Adaline with Stochastic Gradient Descent

- Training an Adaline model with SGD on the iris dataset
  - Training and visualizing a Adaline model using AdalineSGD Class

```
plot_decision_regions(X_std, y, classifier=ada)
plt.title('Adaline - Stochastic Gradient Descent')
plt.xlabel('sepal length [standardized]')
plt.ylabel('petal length [standardized]')
plt.legend(loc='upper left')
plt.show()
```



Submit

- To make sure if you have completed this practice,
  Submit your practice file(Week04_givencode.ipynb) to e-class.
- **Deadline : Saturday 11:59pm**
- Modify your ipynb file name as "Week04_StudentNum_Name.ipynb"
  Ex) **Week04_2020123456_홍길동.ipynb**
- You can upload this file without taking the quiz, but homework will be provided like a quiz every three weeks, so it is recommended to take the quiz as well.

Quiz

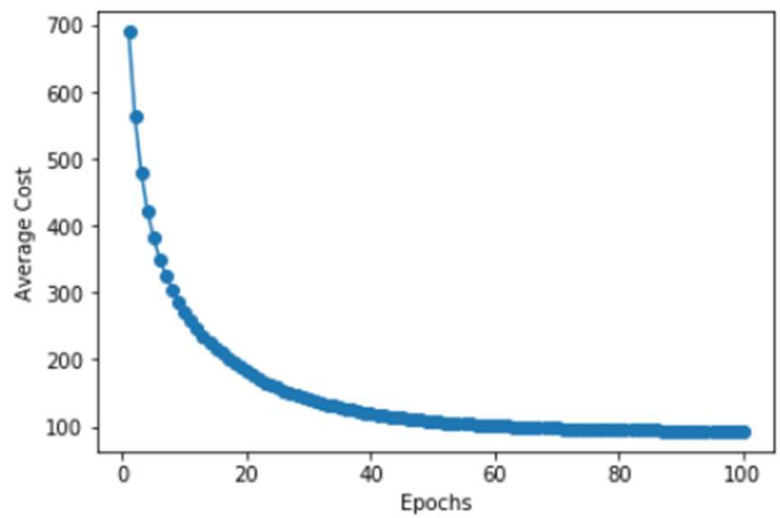
- Training an Adaline model on the Banknote dataset
  - The Banknote dataset(banknote_authentication.csv)
    - Description
    - 0 : Variance of Wavelet Transformed image
    - 1 : Skewness of Wavelet Transformed image
    - 2 : Kurtosis of Wavelet Transformed image
    - 3 : Entropy of image
    - 4 : label(0, 1)
      - 0 : authentic (762 samples)
      - 1 : inauthentic (610 samples)

	0	1	2	3	4
0	3.62160	8.6661	-2.8073	-0.44699	0
1	4.54590	8.1674	-2.4586	-1.46210	0
2	3.86600	-2.6383	1.9242	0.10645	0
3	3.45660	9.5228	-4.0112	-3.59440	0
4	0.32924	-4.4552	4.5718	-0.98880	0

https://machinelearningmastery.com/standard-machine-learning-datasets/

Quiz

- Training an Adaline model on the Banknote dataset
  - Read the dataset into X and y, standardize X, transform y to -1 and 1
  - Using AdalineGD train the model and plot the cost graph.
  - Evaluate the model by computing the accuracy.
  - Hint : Find out what hyperparameter(learning rate : eta) is good for training the model.



https://machinelearningmastery.com/standard-machine-learning-datasets/