



Chapter 2

Perceptron & Adaline

COMP3314
Machine Learning

Outline

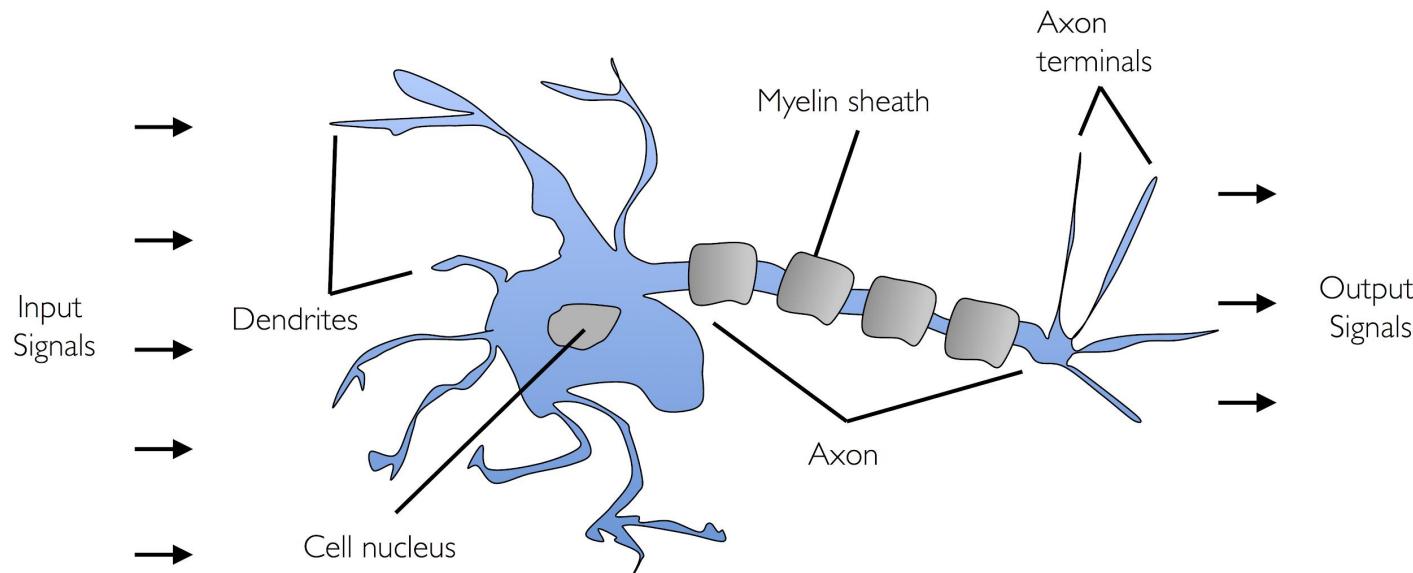
- In this chapter we will implement two of the first published machine learning algorithms for classification
 - Perceptron
 - Adaptive Linear Neurons (Adaline)
- This will lay the groundwork for using more powerful classifiers with the scikit-learn library
- We will
 - Build an intuition for machine learning algorithms
 - Use pandas, NumPy, and Matplotlib to read in, process, and visualize data
 - Implement linear classification algorithms in Python

Outline

- Neuron
- Artificial Neuron
 - History
 - Definition
- Perceptron
 - Perceptron Learning Rule
 - Linearly Separable
 - Implementation
- Adaptive Linear Neuron (Adaline)
 - Implementation
 - Feature Scaling
 - Stochastic Gradient Descent / Mini-Batch Learning

Neuron

- Neurons are interconnected nerve cells in the brain that are involved in the processing and transmitting of chemical and electrical signals



Artificial Neuron - History

- McCulloch and Pitts described the first artificial neuron in 1943 (aka MCP neuron) as a simple logic gate with binary outputs
 - Signals arrive at the dendrites, are integrated into the cell body, and, if the accumulated signal exceeds a certain threshold, an output signal is generated at the axon
- A few years later, in 1958, Rosenblatt published the first concept of the perceptron learning rule based on the MCP neuron
 - Automatically learn the optimal weight coefficients that are then multiplied with the input features

Artificial Neuron - Definition

- Consider a binary classification task where we refer to our two classes as 1 (positive class) and -1 (negative class)
- We can define an activation function

$$\phi(z) = \begin{cases} 1 & \text{if } z \geq \theta \\ -1 & \text{otherwise} \end{cases}$$

where θ is a threshold and z is the net input

$$z = w_1x_1 + \dots + w_m x_m$$

of the input values \mathbf{x} and the corresponding weight vector \mathbf{w}

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

Artificial Neuron - Definition

- For simplicity, we can bring the threshold θ to the left side of the equation and define

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

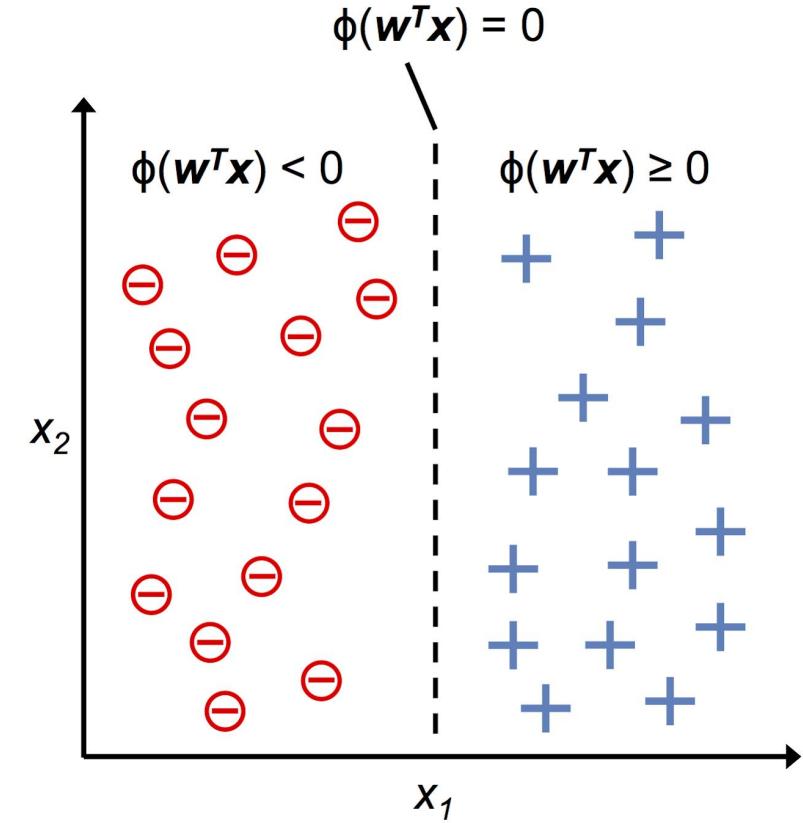
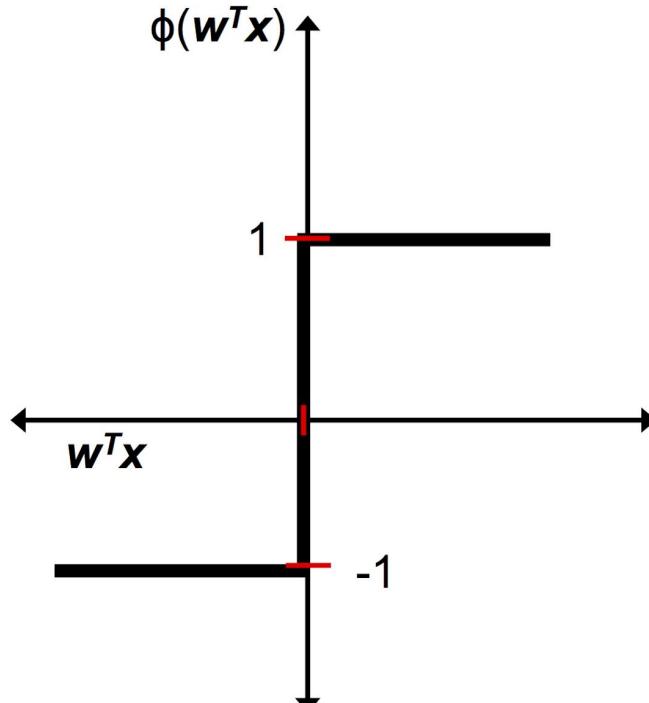
and

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

where $w_0 = -\theta$ and $x_0 = 1$

- w_0 is called the bias unit

Artificial Neuron - Definition



Perceptron Learning Rule

- Rosenblatt's perceptron rule can be summarized by the following steps
 - a. Initialize the weights with small random values
 - b. For each training sample $x^{(i)}$
 - i. Compute the output value \hat{y} ,
i.e. the class label predicted by unit step function
 - i. Update the weights

$$w_j := w_j + \Delta w_j$$

The value of Δw_j is calculated as follows

$$\Delta w_j = \eta \left(y^{(i)} - \hat{y}^{(i)} \right) x_j^{(i)}$$

Where η is the learning rate, $y^{(i)}$ is the true class label of the i th training sample, and $\hat{y}^{(i)}$ is the predicted class label

Weight Update

- All weights in the weight vector are updated simultaneously, which means that we don't recompute the $\hat{y}^{(i)}$ before all of the weights Δw_j are updated
 - I.e., for a two-dimensional dataset, we would write the update as

$$\Delta w_0 = \eta \left(y^{(i)} - output^{(i)} \right)$$

$$\Delta w_1 = \eta \left(y^{(i)} - output^{(i)} \right) x_1^{(i)}$$

$$\Delta w_2 = \eta \left(y^{(i)} - output^{(i)} \right) x_2^{(i)}$$

Weight Update

- In the two scenarios where the perceptron predicts the class label correctly, the weights remain unchanged

$$\Delta w_j = \eta(-1 - (-1)) x_j^{(i)} = 0$$

$$\Delta w_j = \eta(1 - 1) x_j^{(i)} = 0$$

- However, in the case of a wrong prediction, the weights are being pushed towards the direction of the positive or negative target class

$$\Delta w_j = \eta(1 - -1) x_j^{(i)} = \eta(2) x_j^{(i)}$$

$$\Delta w_j = \eta(-1 - 1) x_j^{(i)} = \eta(-2) x_j^{(i)}$$

Weight Update

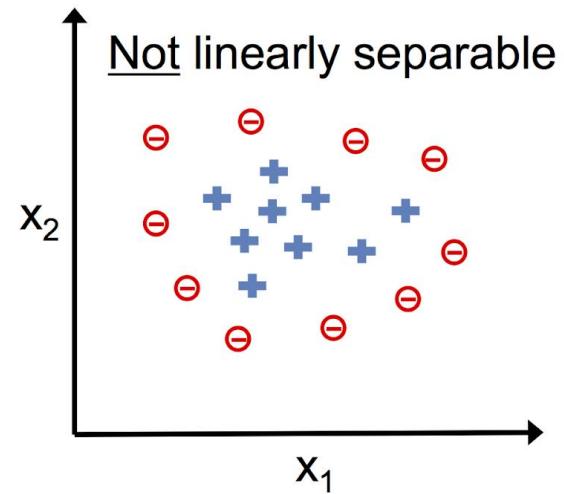
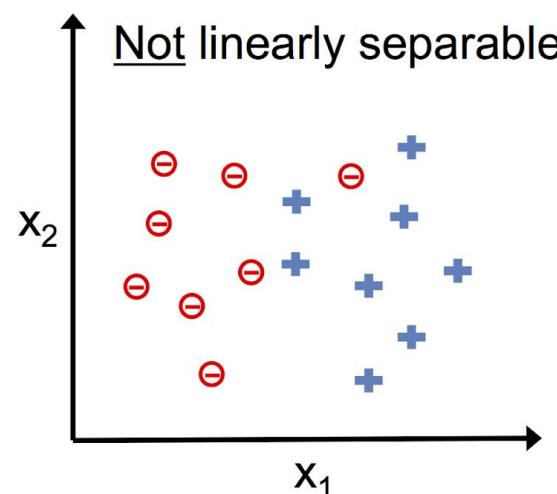
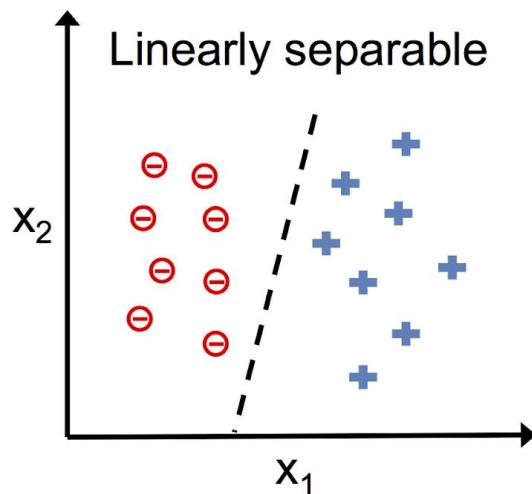
- Note that the weight update is proportional the value of $x_j^{(i)}$

$$\Delta w_j = \eta \left(y^{(i)} - \hat{y}^{(i)} \right) x_j^{(i)}$$

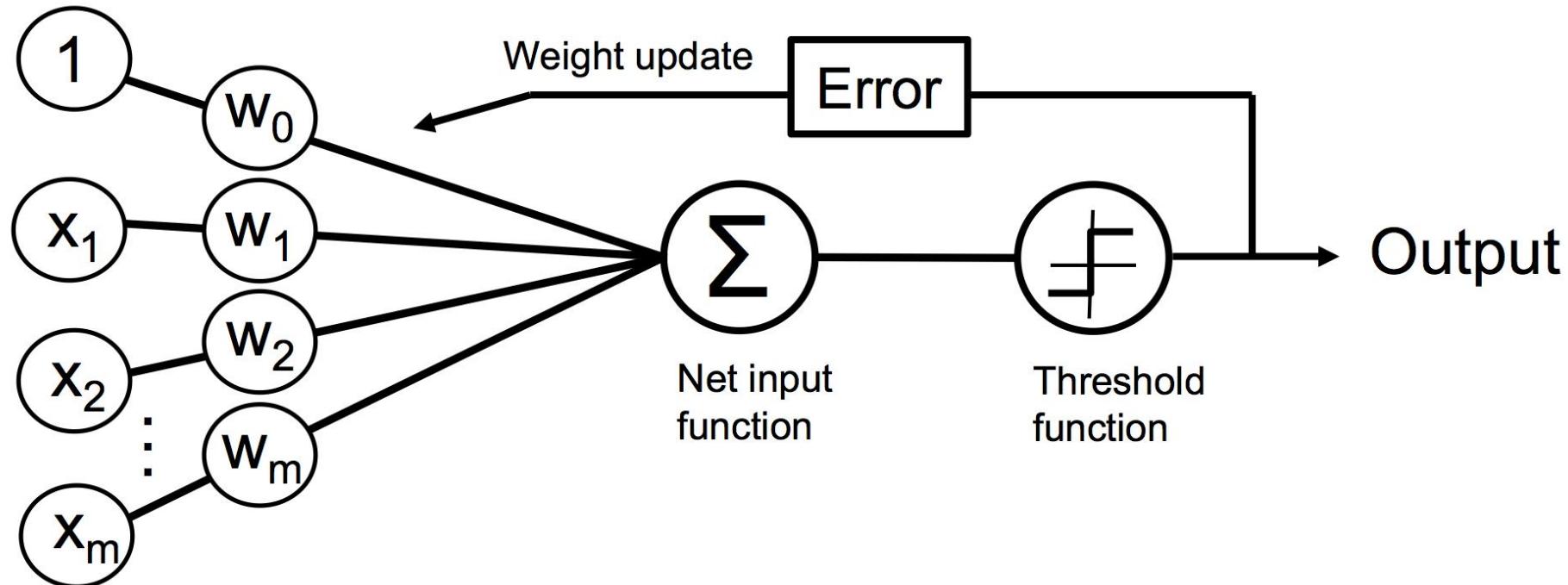
Linearly Separable

- It was shown that the convergence of the perceptron is only guaranteed if the two classes are linearly separable and the learning rate is sufficiently small
- If the two classes can't be separated by a linear decision boundary, we can set a maximum number of passes over the training dataset (epochs) and/or a threshold for the number of tolerated misclassifications
 - The perceptron would never stop updating the weights otherwise

Linearly Separable vs. Not Linearly Separable



Perceptron



Separable Training Samples

Decision Boundary

Before update: $a_1x + b_1y + c_1 = 0$

After update: $a_2x + b_2y + c_2 = 0$

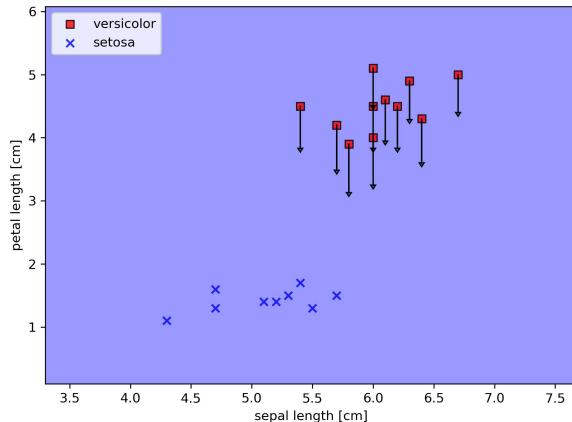
Increasing y-intercept

let $x = 0$ if $-\frac{c_2}{b_2} + \frac{c_1}{b_1} > 0$

Decreasing y-intercept

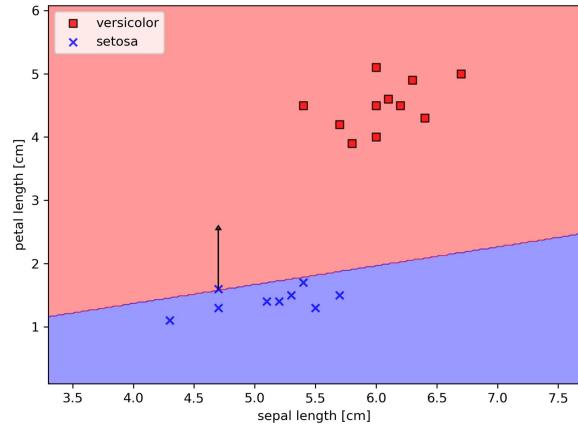
if $-\frac{c_2}{b_2} + \frac{c_1}{b_1} < 0$

Perceptron



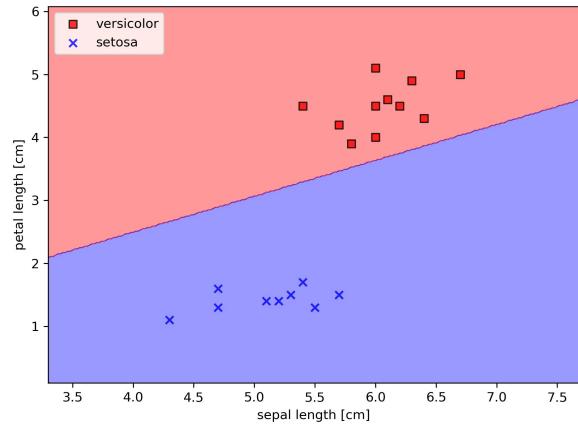
Iteration: 5 Acc: 0.45

Perceptron



Iteration: 10 Acc: 0.95

Perceptron



Iteration: 20 Acc: 1.0

Inseparable Training Samples

Decision Boundary

Before update: $a_1x + b_1y + c_1 = 0$

After update: $a_2x + b_2y + c_2 = 0$

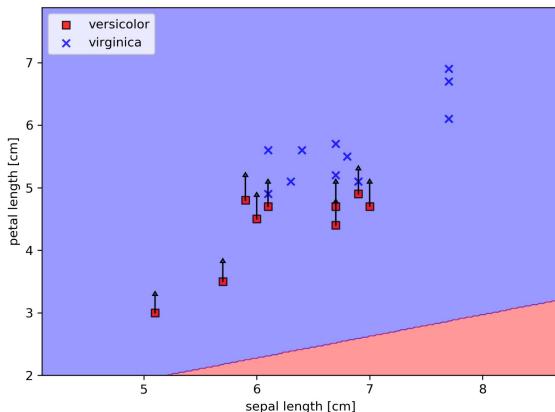
Increasing y-intercept

let $x = 0$ if $-\frac{c_2}{b_2} + \frac{c_1}{b_1} > 0$

Decreasing y-intercept

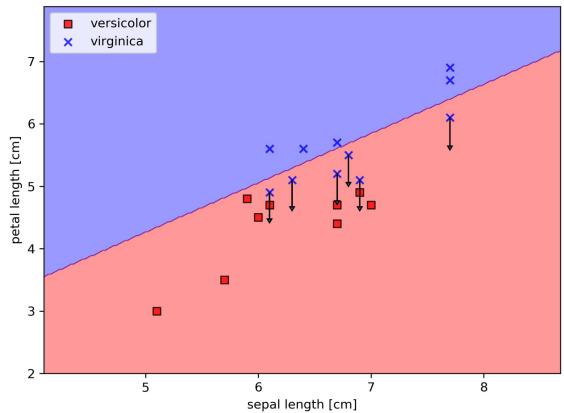
if $-\frac{c_2}{b_2} + \frac{c_1}{b_1} < 0$

Perceptron



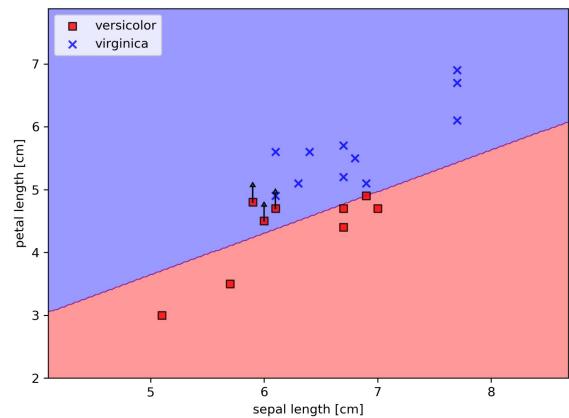
Iteration: 5 Acc: 0.55

Perceptron



Iteration: 10 Acc: 0.7

Perceptron



Iteration: 20 Acc: 0.85

Iris Dataset - Loading

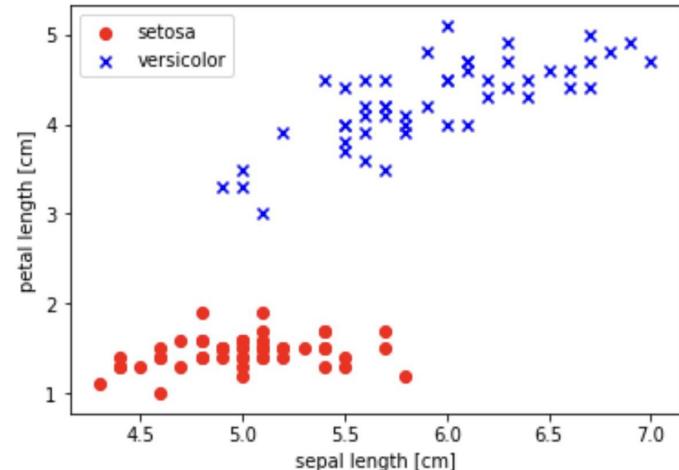
```
import pandas as pd
df = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data', header=None)
df.head()
```

	0	1	2	3	4
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa

Pandas dataframes is a data format that pandas is using.
You may think of it as a spreadsheet.
Columns have potentially different data types.

Iris Dataset - Preprocessing and Plotting

```
import matplotlib.pyplot as plt
import numpy as np
y = df.iloc[0:100, 4].values
y = np.where(y == 'Iris-setosa', -1, 1)
X = df.iloc[0:100, [0, 2]].values
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()
```



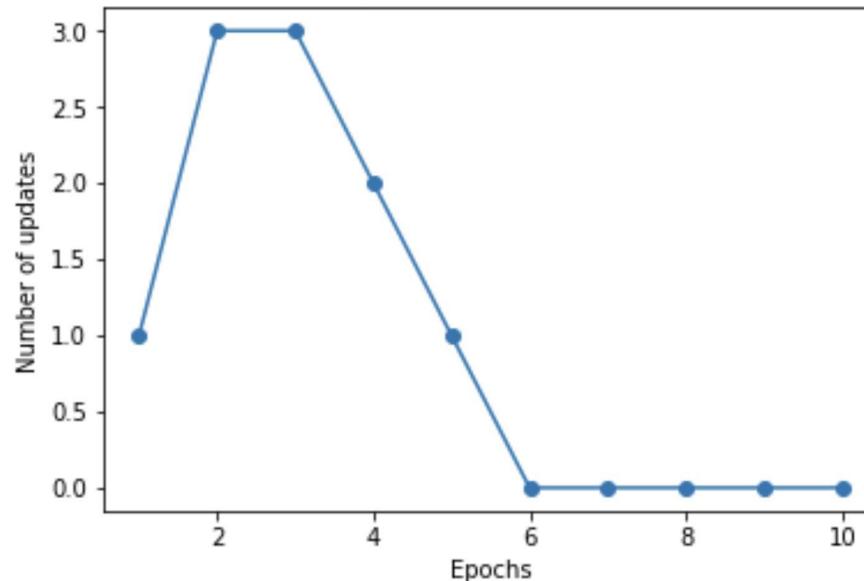
Perceptron - Implementation

```
import numpy as np
class Perceptron(object):
    def __init__(self, eta=0.01, n_iter=50, random_state=1):
        self.eta = eta
        self.n_iter = n_iter
        self.random_state = random_state
    def fit(self, X, y):
        rgen = np.random.RandomState(self.random_state)
        self.w_ = rgen.normal(loc=0.0, scale=0.01, size=1 + X.shape[1])
        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):
        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)
```

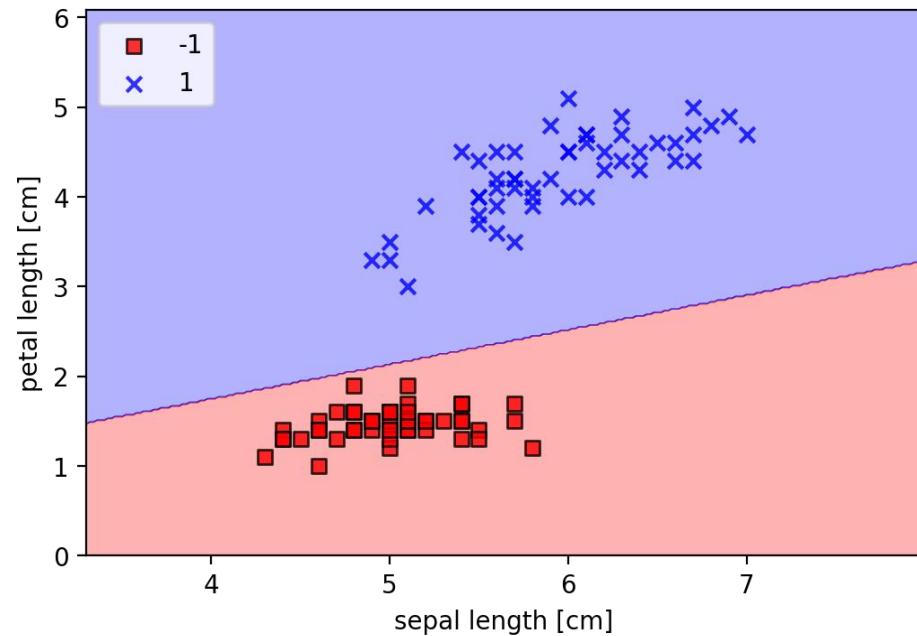
$$\Delta w_j = \eta \left(y^{(i)} - \hat{y}^{(i)} \right) x_j^{(i)}$$

Perceptron - Training

```
ppn = Perceptron(eta=0.1, n_iter=10)
ppn.fit(X, y)
plt.plot(range(1, len(ppn.errors_) + 1), ppn.errors_, marker='o')
plt.xlabel('Epochs')
plt.ylabel('Number of updates')
plt.show()
```



Perceptron - Plotting Decision Region



```
from matplotlib.colors import ListedColormap
def plot_decision_regions(X, y, classifier, resolution=0.02):
    markers = ('s', 'x', 'o', '^', 'v')
    colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
    cmap = ListedColormap(colors[:len(np.unique(y))])
    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))
    Z = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
    Z = Z.reshape(xx1.shape)
    plt.contourf(xx1, xx2, Z, alpha=0.3, cmap=cmap)
    plt.xlim(xx1.min(), xx1.max())
    plt.ylim(xx2.min(), xx2.max())
    for idx, cl in enumerate(np.unique(y)):
        plt.scatter(x=X[y == cl, 0], y=X[y == cl, 1], alpha=0.8,
                    c=colors[idx], marker=markers[idx],
                    label=cl, edgecolor='black')
plot_decision_regions(X, y, classifier=ppn)
plt.xlabel('sepal length [cm]')
plt.ylabel('petal length [cm]')
plt.legend(loc='upper left')
plt.show()
```

Outline

- Neuron
- Artificial Neuron
 - History
 - Definition
- Perceptron
 - Perceptron Learning Rule
 - Linearly Separable
 - Implementation
- Adaptive Linear Neuron (Adaline)
 - Implementation
 - Feature Scaling
 - Stochastic Gradient Descent / Mini-Batch Learning

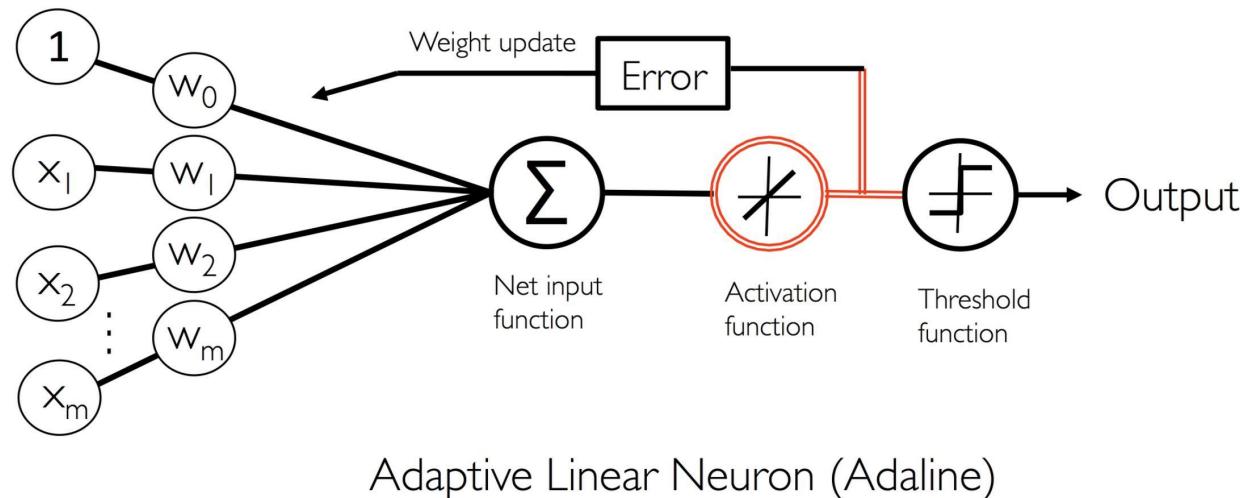
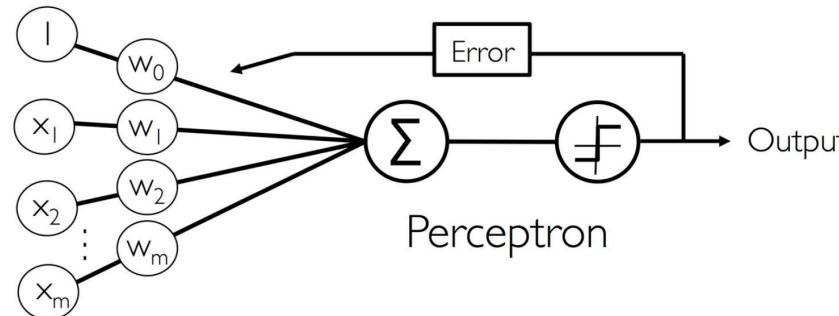
Adaptive Linear Neuron

- ADaptive LInear NEuron: Adaline
- Improvement on Perceptron algorithm
- Published in 1960 by Bernard Widrow and Ted Hoff
- In Adaline the weights are updated based on a linear activation function

$$\phi(w^T x) = w^T x$$

- While the linear activation function is used for learning the weights, we still use a threshold function to make the final prediction

Adaptive Linear Neuron



Objective Function

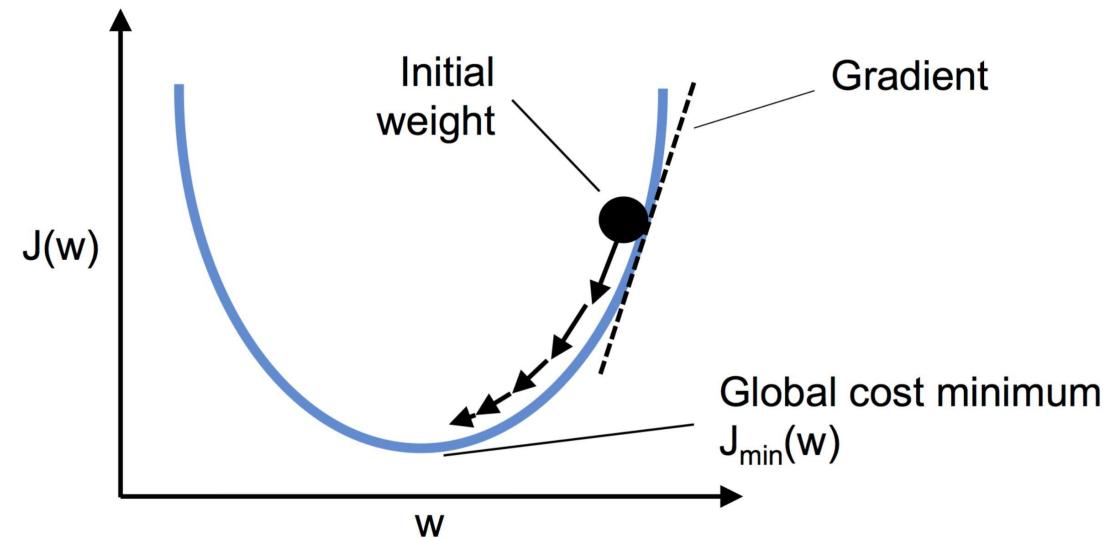
- One of the key ingredients of supervised ML algorithms is the objective function that is to be optimized
 - E.g., cost function that we want to minimize
- In Adaline the cost function J is defined as the sum of squared errors (SSE) between the calculated and true class label

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

- It can be shown that J is differentiable and convex
 - It will be easy to minimize (using e.g. gradient descent)

Gradient Descent

- A generic optimization algorithm
 - Capable of finding optimal solutions to a wide range of problems
- Main idea
 - Tweak parameters iteratively in order to minimize a cost function
 - Measures the local gradient of the cost function with regard to the parameter vector and goes in the direction of descending gradient
 - Once the gradient is zero, it has reached a minimum



Weight Update

- Update the weights by taking a step in the opposite direction of the gradient $\nabla J(\mathbf{w})$ of our cost function $J(\mathbf{w})$

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

- The weight change $\Delta\mathbf{w}$ is defined as the negative gradient multiplied by the learning rate η

$$\Delta\mathbf{w} = -\eta \nabla J(\mathbf{w})$$

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

$$= \frac{1}{2} \frac{\partial}{\partial w_j} \sum_i \left(y^{(i)} - \phi(z^{(i)}) \right)^2$$

$$= \frac{1}{2} \sum_i 2 \left(y^{(i)} - \phi(z^{(i)}) \right) \frac{\partial}{\partial w_j} \left(y^{(i)} - \phi(z^{(i)}) \right)$$

$$= \sum_i \left(y^{(i)} - \phi(z^{(i)}) \right) \frac{\partial}{\partial w_j} \left(y^{(i)} - \sum_k \left(w_k x_k^{(i)} \right) \right)$$

$$= \sum_i \left(y^{(i)} - \phi(z^{(i)}) \right) \left(-x_j^{(i)} \right)$$

$$= - \sum_i \left(y^{(i)} - \phi(z^{(i)}) \right) x_j^{(i)}$$

Weight Update

- To compute the gradient of the cost function, we compute the partial derivative of the cost function with respect to each weight w_j

$$\frac{\partial J}{\partial w_j} = -\sum_i \left(y^{(i)} - \phi(z^{(i)}) \right) x_j^{(i)}$$

- The update of weight w_j can then be written as

$$\Delta w_j = -\eta \frac{\partial J}{\partial w_j} = \eta \sum_i \left(y^{(i)} - \phi(z^{(i)}) \right) x_j^{(i)}$$

- Batch gradient descent
 - Note that the weight update is calculated based on all samples in the training set (instead of updating the weights incrementally after each sample)

Adaline - Implementation

```

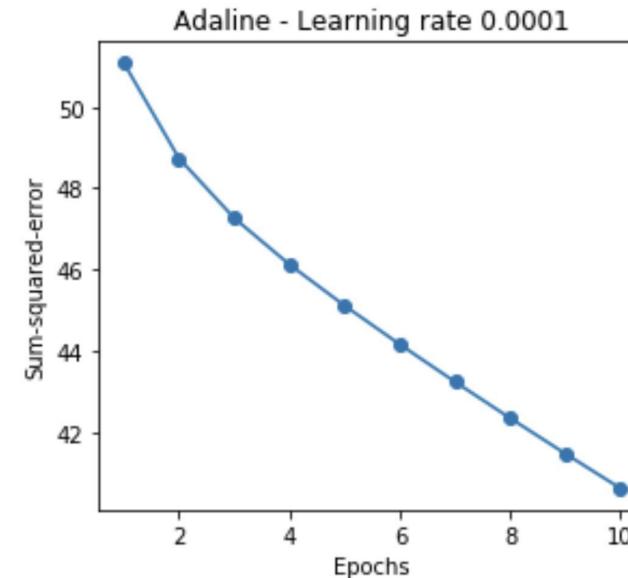
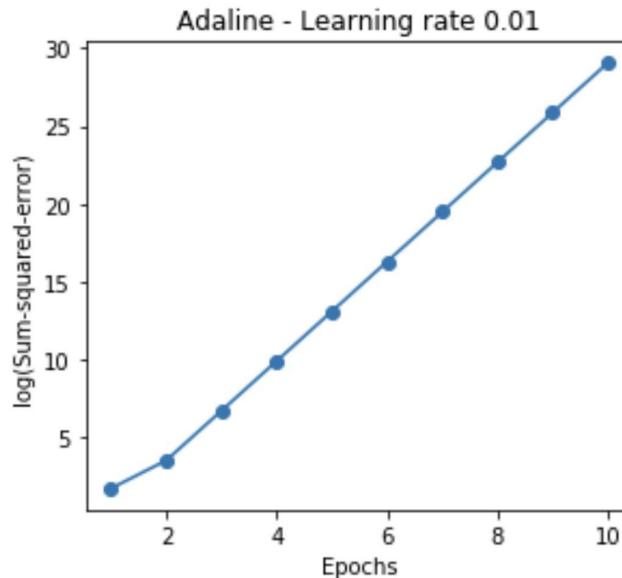
class AdalineGD(object):
    def __init__(self, eta=0.01, n_iter=50, random_state=1):
        self.eta = eta
        self.n_iter = n_iter
        self.random_state = random_state
    def fit(self, X, y):
        rgen = np.random.RandomState(self.random_state)
        self.w_ = rgen.normal(loc=0.0, scale=0.01, size=1 + X.shape[1])
        self.cost_ = []
        for i in range(self.n_iter):
            net_input = self.net_input(X)
            output = self.activation(net_input)
            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
    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)

```

$$\Delta w_j = -\eta \frac{\partial J}{\partial w_j} = \eta \sum_i (y^{(i)} - \phi(z^{(i)})) x_j^{(i)}$$

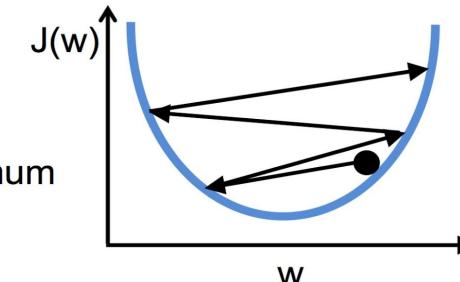
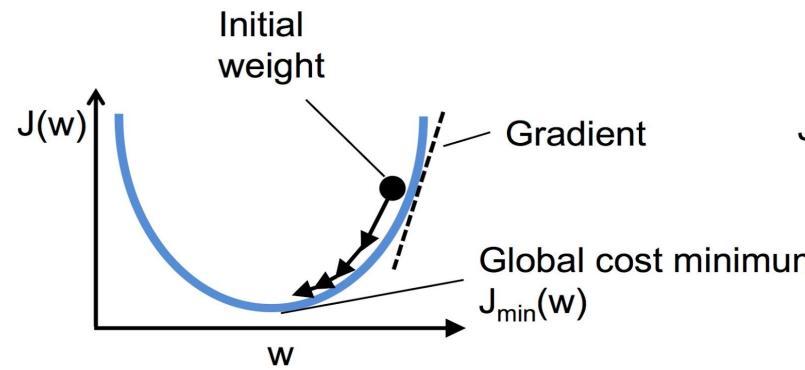
$$J(\mathbf{w}) = \frac{1}{2} \sum_i (y^{(i)} - \phi(z^{(i)}))^2$$

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

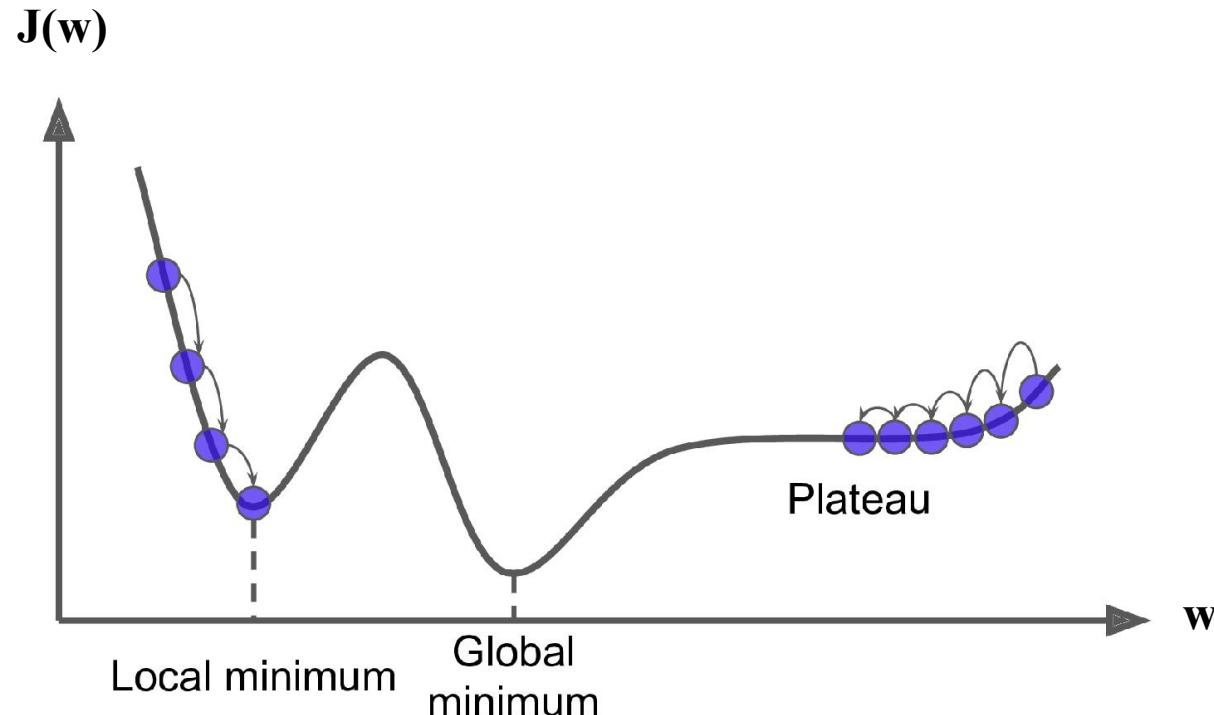


Learning Rate

- Important hyperparameter of Gradient Descent
- If too small
 - Requires excessive number of iterations to converge
- If too large
 - Might jump across the valley and end up on the other side
 - Algorithm may diverge, with larger and larger values

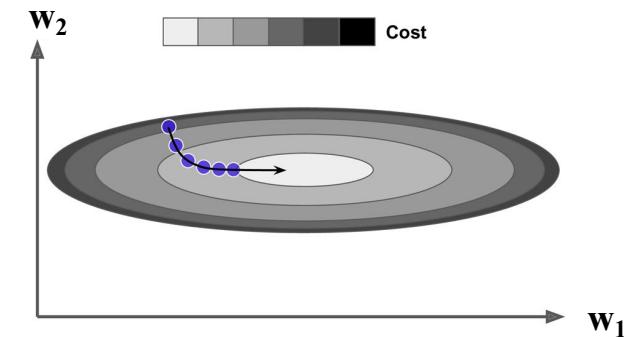
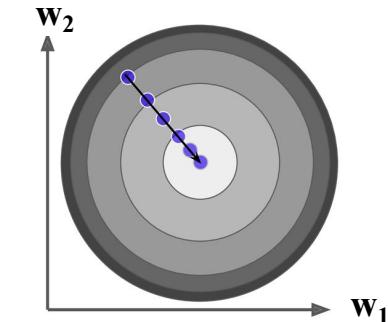


Gradient Descent Pitfalls



Feature Scaling

- Many ML algorithms require feature scaling for optimal performance
 - E.g., gradient descent converges more quickly if our data follows a standard distribution
- Standardization
 - A feature scaling method
 - After standardization, feature have
 - a mean value of 0
 - a standard deviation of 1



Standardization

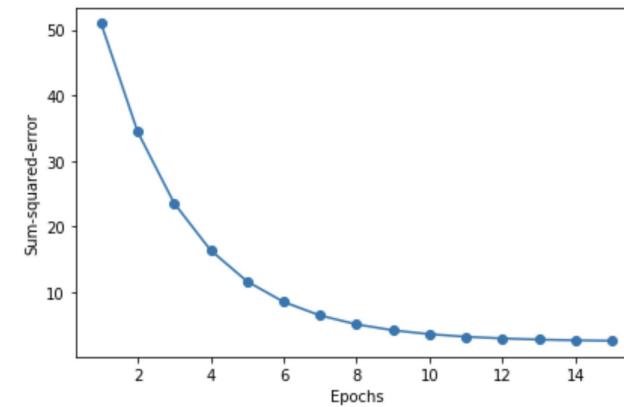
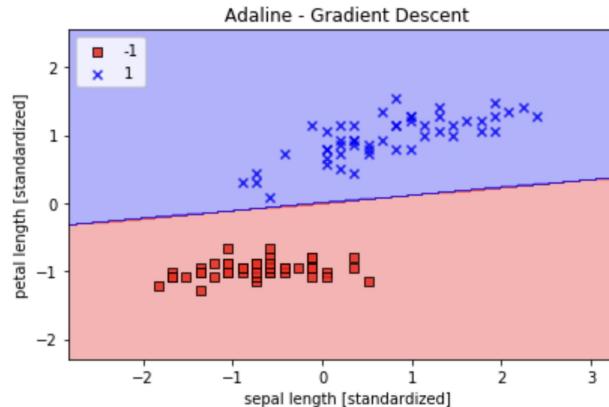
- For instance, to standardize the j th feature, we can simply subtract the sample mean μ_j from every training sample and divide it by its standard deviation σ_j

$$\mathbf{x}'_j = \frac{\mathbf{x}_j - \mu_j}{\sigma_j}$$

- Here, \mathbf{x}_j is a vector consisting of the j th feature values of all training samples n
- This standardization technique is applied to each feature j in our dataset

```
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()
```

```
ada = AdalineGD(n_iter=15, eta=0.01)
ada.fit(X_std, y)
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.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.show()
```



Separable Training Samples

Decision Boundary

Before update: $a_1x + b_1y + c_1 = 0$

After update: $a_2x + b_2y + c_2 = 0$

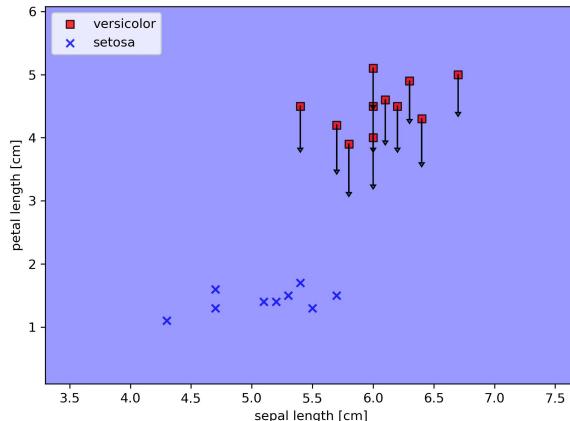
Increasing y-intercept

let $x = 0$ if $-\frac{c_2}{b_2} + \frac{c_1}{b_1} > 0$

Decreasing y-intercept

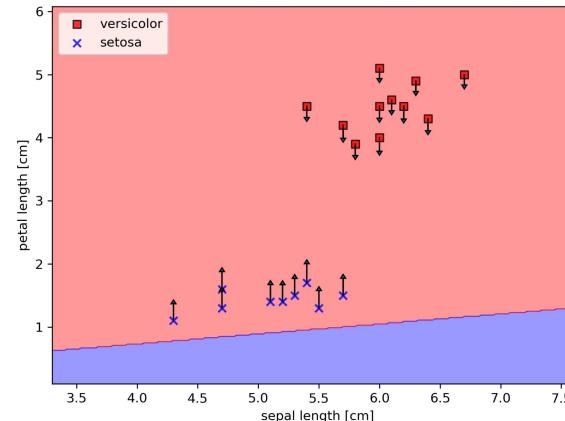
if $-\frac{c_2}{b_2} + \frac{c_1}{b_1} < 0$

Perceptron



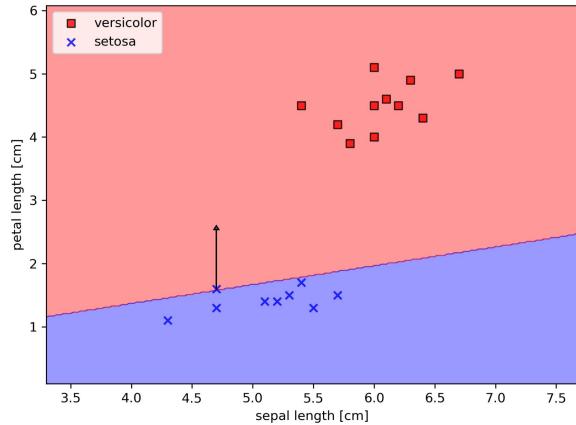
Iteration: 5 Acc: 0.45

Adaline



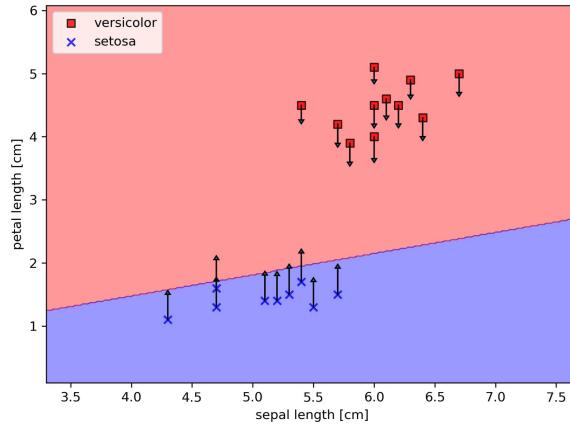
Iteration: 5 Acc: 0.55

Perceptron



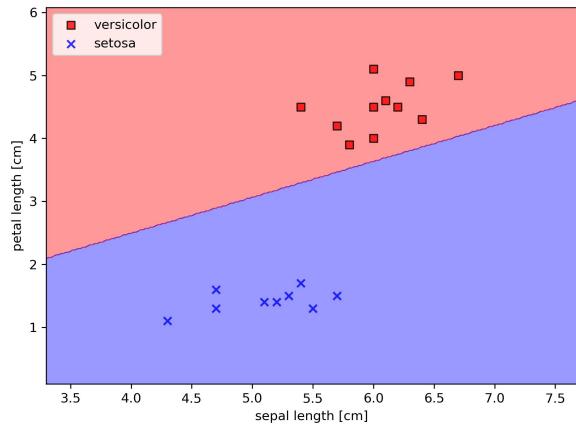
Iteration: 10 Acc: 0.95

Adaline



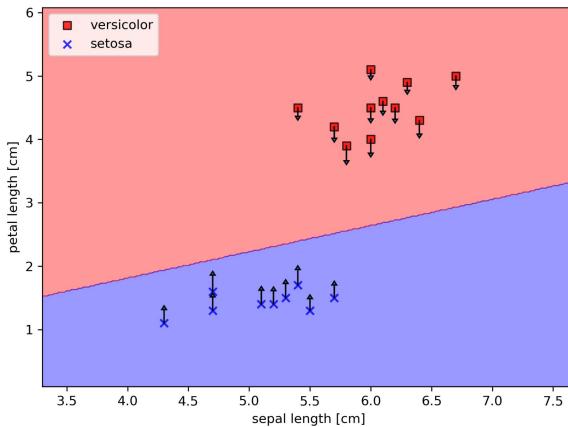
Iteration: 10 Acc: 1.0

Perceptron



Iteration: 20 Acc: 1.0

Adaline



Iteration: 20 Acc: 1.0

Inseparable Training Samples

Decision Boundary

Before update: $a_1x + b_1y + c_1 = 0$

After update: $a_2x + b_2y + c_2 = 0$

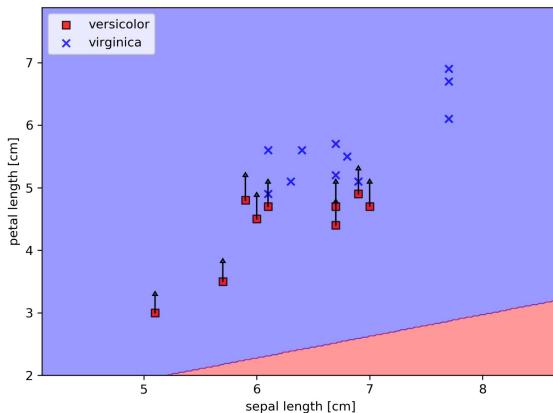
Increasing y-intercept

let $x = 0$ if $-\frac{c_2}{b_2} + \frac{c_1}{b_1} > 0$

Decreasing y-intercept

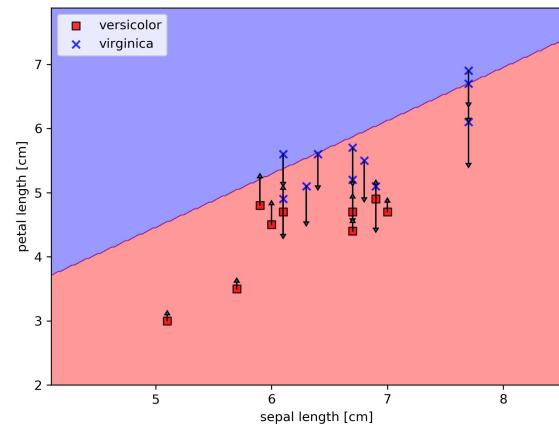
if $-\frac{c_2}{b_2} + \frac{c_1}{b_1} < 0$

Perceptron



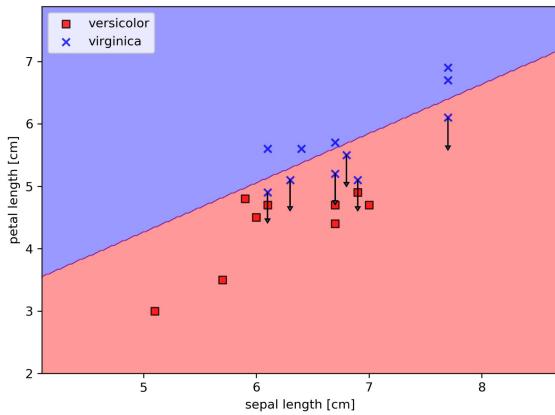
Iteration: 5 Acc: 0.55

Adaline



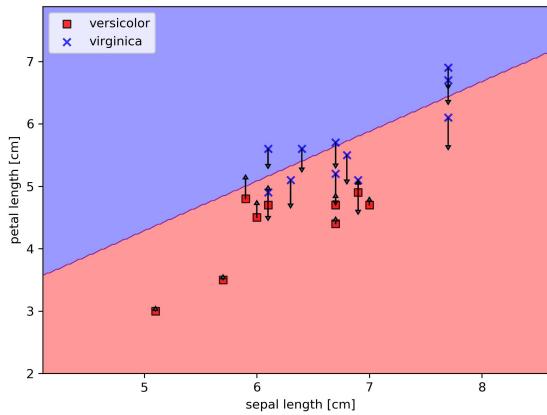
Iteration: 5 Acc: 0.55

Perceptron



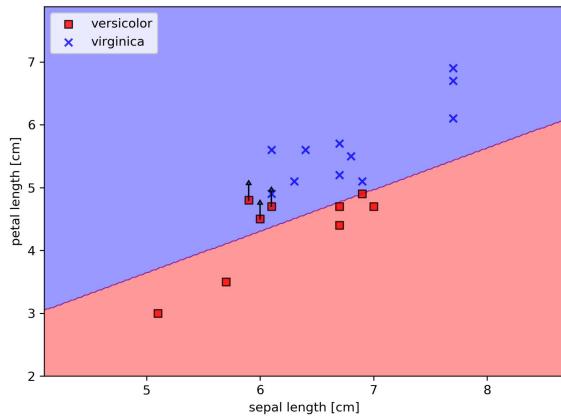
Iteration: 10 Acc: 0.7

Adaline



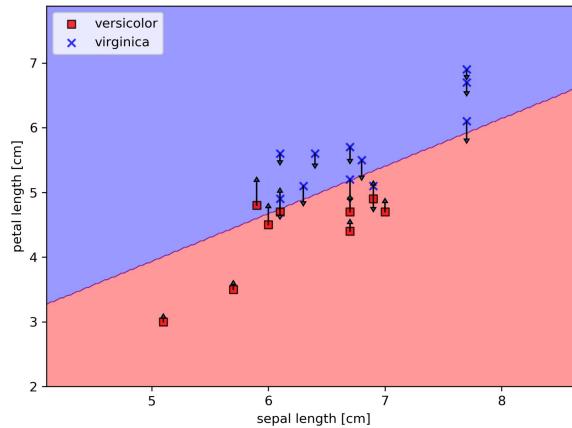
Iteration: 10 Acc: 0.7

Perceptron



Iteration: 20 Acc: 0.85

Adaline



Iteration: 20 Acc: 0.9

Stochastic Gradient Descent

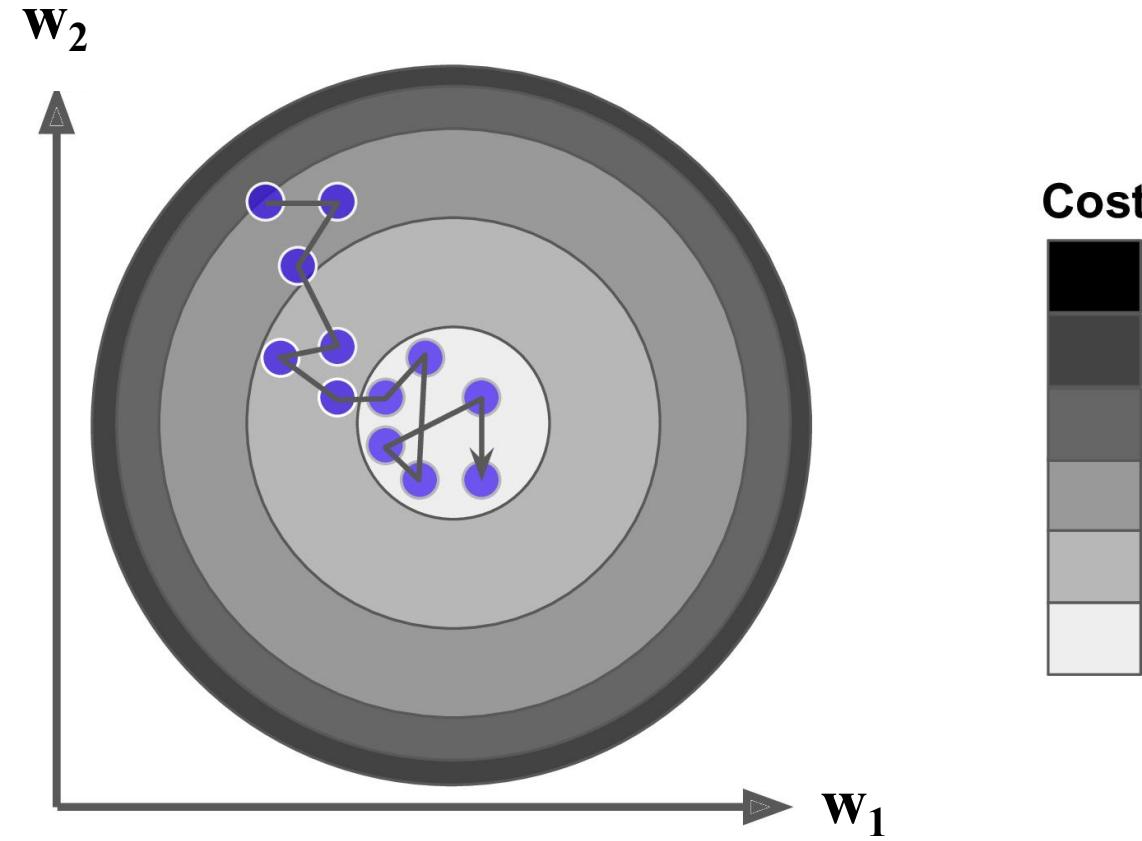
- Imagine we have a very large dataset with millions of data points
 - Not uncommon in ML applications
 - Running batch gradient descent can be computationally costly in such scenarios since we need to reevaluate the whole training dataset each time we take one step
- Stochastic Gradient Descent is a popular alternative
 - Instead of updating the weights based on the sum of the accumulated errors over all samples $\mathbf{x}^{(i)}$

$$\Delta \mathbf{w} = \eta \sum_i \left(y^{(i)} - \phi(z^{(i)}) \right) \mathbf{x}^{(i)}$$

we update all the weights incrementally for each training sample

Stochastic Gradient Descent

- Typically reaches convergences faster because of the more frequent weight updates
- Can escape shallow local minima more readily if we are working with nonlinear cost functions
- It is important to present it training data in a random order
- In addition shuffle the training set for every epoch to prevent cycles
- Another advantage of stochastic gradient descent is that we can use it for online learning
 - In online learning, our model is trained on the fly as new training data arrives



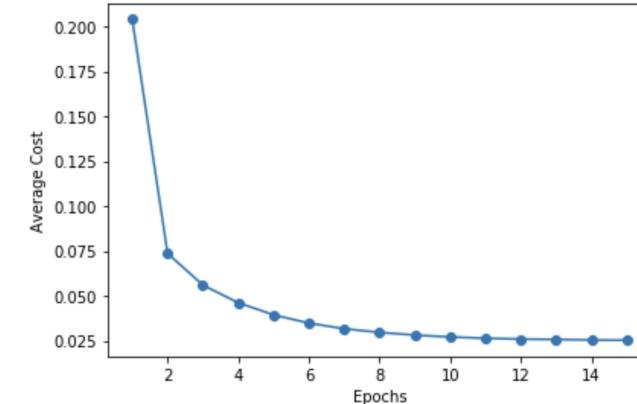
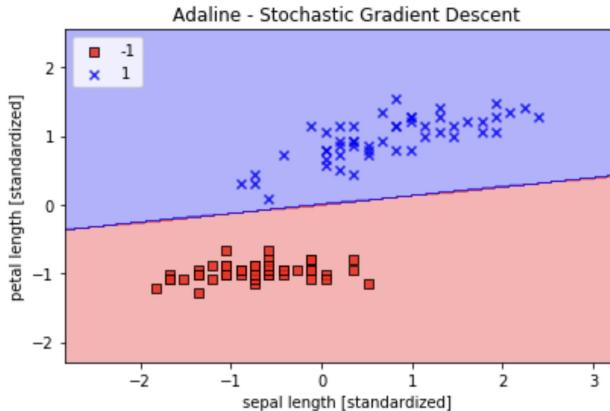
Mini-Batch Learning

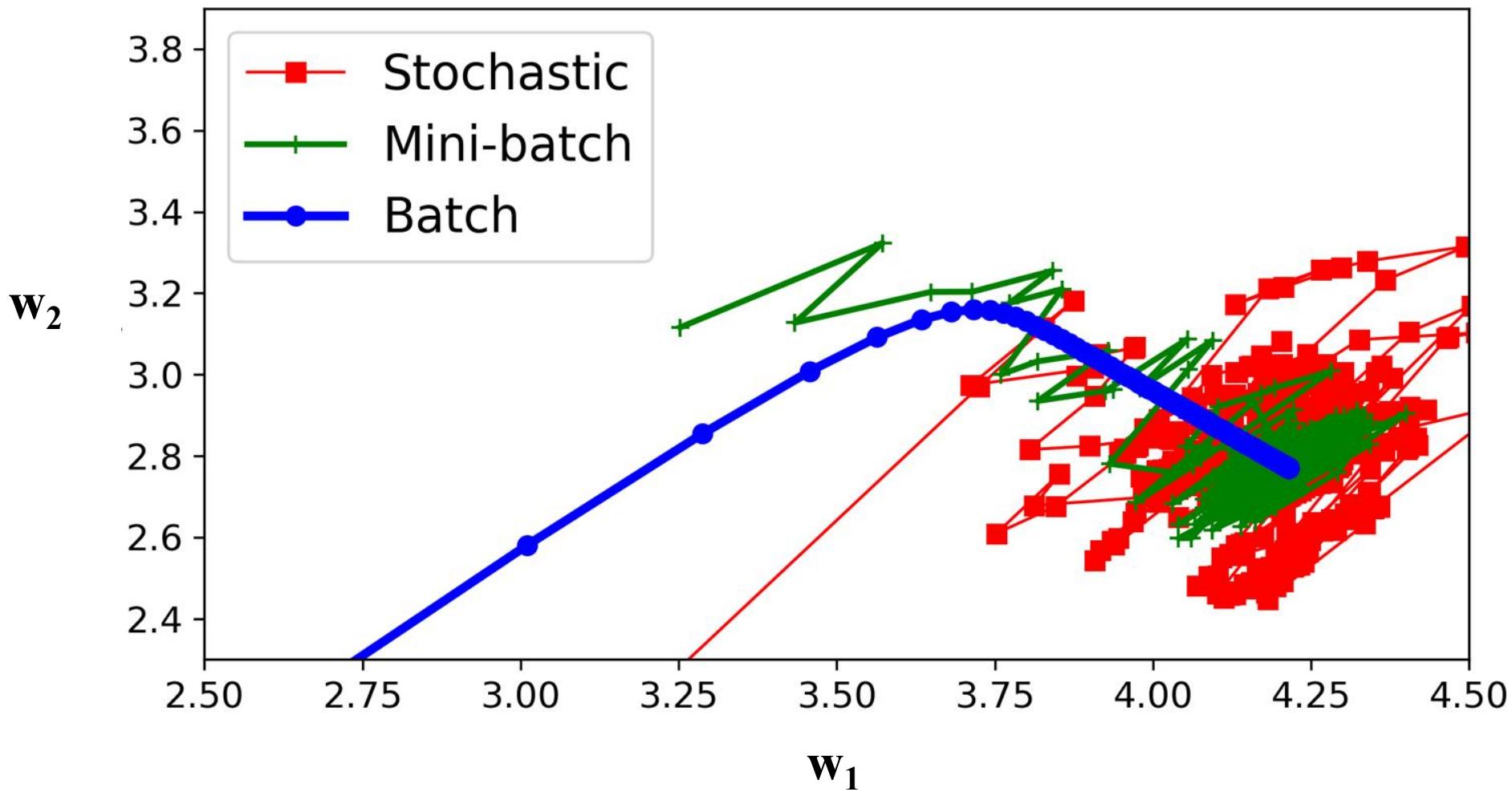
- A compromise between batch gradient descent and stochastic gradient descent
- Apply batch gradient descent to smaller subsets of the training data
 - E.g., 32 samples at a time
- The advantage over batch gradient descent is that convergence is reached faster via mini-batches because of the more frequent weight updates
- Furthermore, mini-batch learning allows us to replace the for loop over the training samples in stochastic gradient descent with vectorized operations

```
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
    def fit(self, X, y):
        self._initialize_weights(X.shape[1])
        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
    def partial_fit(self, X, y):
        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):
    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)
```

```
ada = AdalineSGD(n_iter=15, eta=0.01, random_state=1)
ada.fit(X_std, y)
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.tight_layout()
plt.show()
plt.plot(range(1, len(ada.cost_) + 1), ada.cost_, marker='o')
plt.xlabel('Epochs')
plt.ylabel('Average Cost')
plt.tight_layout()
plt.show()
```





Scikit-learn Library

- Previously we implemented the perceptron rule in Python ourselves
- We will now use the scikit-learn library and train a perceptron model similar to the one we implemented
 - This will serve as an intro to the scikit-learn library

Iris Dataset

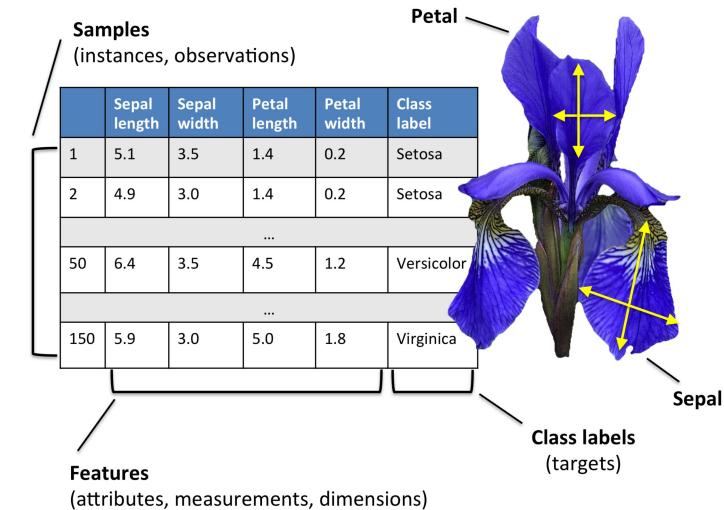
- Conveniently, the Iris dataset is already available via scikit-learn

```
from sklearn import datasets
iris = datasets.load_iris()
```

- We will only use two features (petal length and petal width) from the Iris dataset for visualization purposes
- Assign all flower samples to the feature matrix \mathbf{X} and the corresponding class labels of the flower species to the vector \mathbf{y}

```
x = iris.data[:, [2, 3]]
y = iris.target
import numpy as np
print('Class labels:', np.unique(y))
```

Class labels: [0 1 2]



Training and Testing

- To evaluate how well a trained model performs on unseen data, we will further split the dataset into separate training and test datasets

```
from sklearn.model_selection import train_test_split  
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,  
                                                random_state=1, stratify=y)
```

- Note that the `train_test_split` function shuffles the training sets internally and performs stratification before splitting
 - Otherwise, all class 0 and class 1 samples would have ended up in the training set, and the test set would consist of 45 samples from class 2

Stratification

```
from sklearn.model_selection import train_test_split  
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,  
                                                 random_state=1, stratify=y)
```

- We took advantage of the built-in support for stratification
- In this context, stratification means that the `train_test_split` method returns training and test subsets that have the same proportions of class labels as the input dataset

```
print('Labels counts in y:', np.bincount(y))  
print('Labels counts in y_train:', np.bincount(y_train))  
print('Labels counts in y_test:', np.bincount(y_test))
```

```
Labels counts in y: [50 50 50]  
Labels counts in y_train: [35 35 35]  
Labels counts in y_test: [15 15 15]
```

Multiclass Classification

- n (number of classes) > 2
- Some classification algorithms naturally permit $n > 2$
 - Others are by nature binary algorithms
- OvA or One-versus-Rest (OvR)
 - Train one classifier per class, where the particular class is treated as the positive class
 - Samples from all other classes are considered negative classes
- If we were to classify a new data sample, we would use our n classifiers, and assign the class label with the largest classifier output to the particular sample

Feature Scaling

- Recall that many machine learning and optimization algorithms also require feature scaling for optimal performance
- Here, we will standardize the features using the StandardScaler class from scikit-learn's preprocessing module

```
from sklearn.preprocessing import StandardScaler  
sc = StandardScaler()  
sc.fit(X_train)  
X_train_std = sc.transform(X_train)  
X_test_std = sc.transform(X_test)
```

Training

- We can now train a perceptron model
- Most algorithms in scikit-learn already support multiclass classification by default via the One-versus-Rest (OvR) method, which allows us to feed the three flower classes to the perceptron all at once

```
from sklearn.linear_model import Perceptron
ppn = Perceptron(eta0=0.1, random_state=1)
ppn.fit(X_train_std, y_train)
y_pred = ppn.predict(X_train_std)
print('Misclassified training samples:', (y_train!=y_pred).sum())

Misclassified training samples: 6
```

Scikit-Learn Help

- The Perceptron, as well as other scikit-learn functions and classes, often have additional parameters that we omit for clarity
- You can read more about those parameters using the help function

```
help(Perceptron)

Help on class Perceptron in module sklearn.linear_model._perceptron:

class Perceptron(sklearn.linear_model._stochastic_gradient.BaseSGDClassifier)
    Perceptron

    Read more in the :ref:`User Guide <perceptron>`.

    Parameters
    -----
    penalty : {'l2','l1','elasticnet'}, default=None
        The penalty (aka regularization term) to be used.

    alpha : float, default=0.0001
```

Testing

- Having trained a model in scikit-learn, we can make predictions via the predict method on the test data

```
y_pred = ppn.predict(X_test_std)
print('Misclassified samples:', (y_test != y_pred).sum())
```

```
Misclassified samples: 1
```

Testing - Accuracy

- The scikit-learn library also implements a large variety of different performance metrics that are available via the metrics module
 - We can calculate the classification accuracy as follows

```
from sklearn.metrics import accuracy_score
print('Accuracy: %.3f' % accuracy_score(y_test, y_pred))
```

```
Accuracy: 0.978
```

- Alternatively, each classifier in scikit-learn has a score method

```
print('Accuracy: %.3f' % ppn.score(X_test_std, y_test))
```

```
Accuracy: 0.978
```

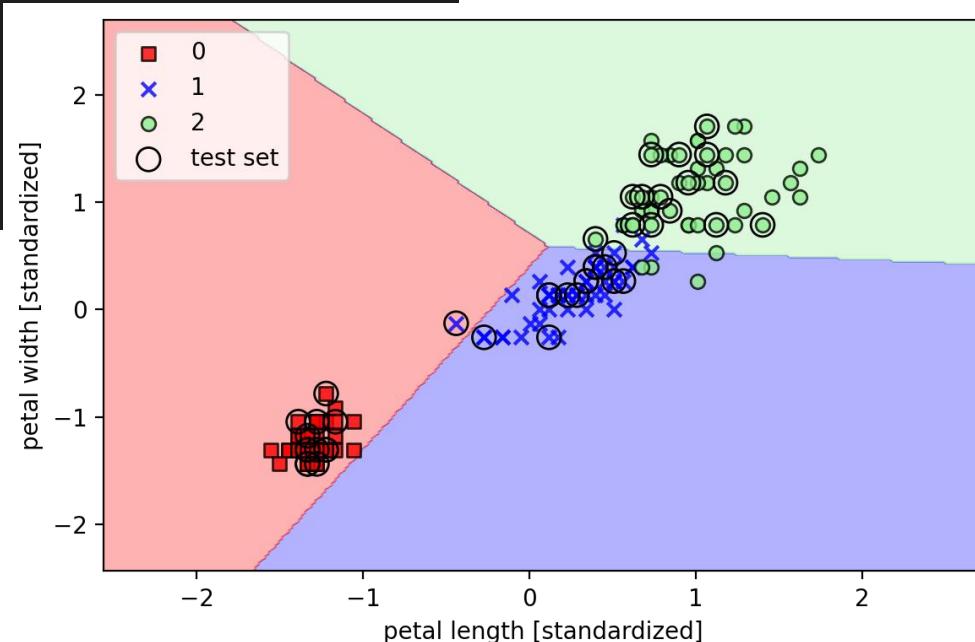
Decision Regions Plotting

- Finally, we can use our `plot_decision_regions` function to plot the decision regions of our newly trained perceptron model
 - Visualize how well it separates the different flower samples
- However, we need to add a small modification to highlight the samples from the test dataset via small circles

Decision Regions Plotting

Decision Regions Plotting

```
X_combined_std = np.vstack((X_train_std, X_test_std))
y_combined = np.hstack((y_train, y_test))
plot_decision_regions(X=X_combined_std, y=y_combined,
                      classifier=ppn, test_idx=range(105, 150))
plt.xlabel('petal length [standardized]')
plt.ylabel('petal width [standardized]')
plt.legend(loc='upper left')
plt.tight_layout()
plt.show()
```

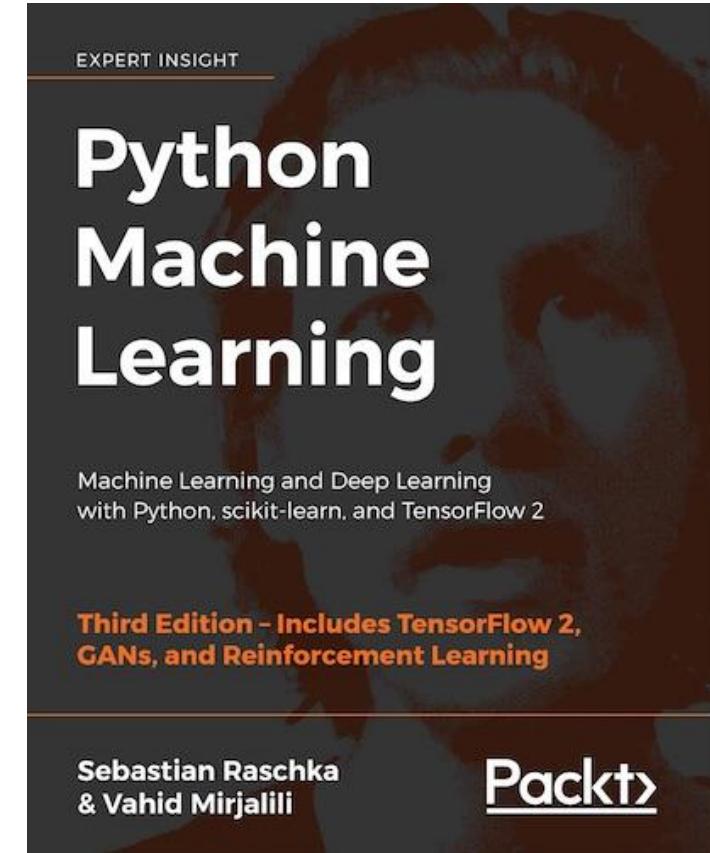


Summary

- Gained a good understanding of the basic concepts of linear classifiers for supervised learning
- Implemented
 - Perceptron
 - Adaline
- Efficient training via a vectorized implementation of gradient descent and online learning via stochastic gradient descent

References

- Most materials in this chapter are based on
 - [Book](#)
 - [Code](#)



References

- Some materials in this chapter are based on
 - [Book](#)
 - [Code](#)

