Adaline (Adaptive linear neurons and the convergence of learning)

August 9, 2020

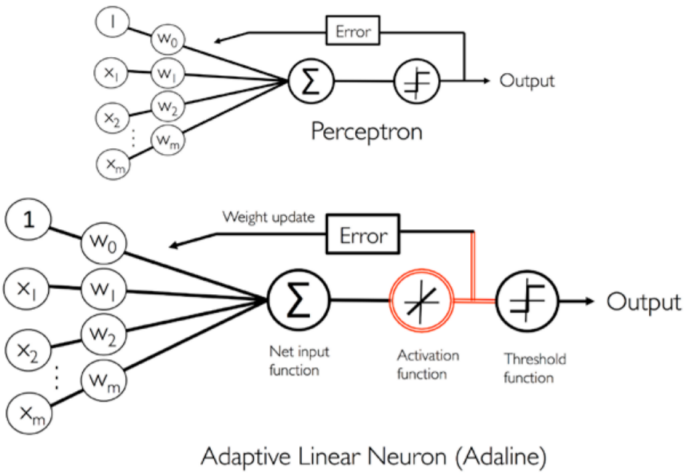
1 Adaptive linear neurons

1.1 Overview

Adaline được công bố bởi Bernard Widrow và nghiên cứu sinh Tedd Hoff của ông ấy chỉ sau 1 vài năm so với thuật toán perceptron của Rosenblatt. Thuật toán này đặc biệt thú vị bởi nó minh họa các khái niệm chính để xác định và giảm các hàm chi phí liên tục. Điều này đặt nền tảng cho việc hiểu các thuật toán học máy nâng cao cho việc phân loại như: logistic regression, support vector machines, and regression models. Sự khác biệt giữa Adaline và Perceptron là weights được cập nhật dựa trên linear activation function chứ không phải là 1 unit step function như trong Perceptron. Trong Adaptive thì linear activation function ( ), đơn giản là định nghĩa hàm của net input, vậy nên:

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

Trong khi linear activation function dùng để học weights thì chúng ta vẫn dùng threshold function để dự đoán điểm kết thúc. Hàm này thì giống với unit step function đã được đề cập trước đó của Perceptron.



Giải thích về cơ chế predict cho 2 hình trên:

1. Perceptron:

Step 1: Inner product giữa một vector đầu vào (thứ cần dc classified) và các model params

Step 2: Dùng threshold func, thường là sgn (hình vẽ cũng vẽ sgn)

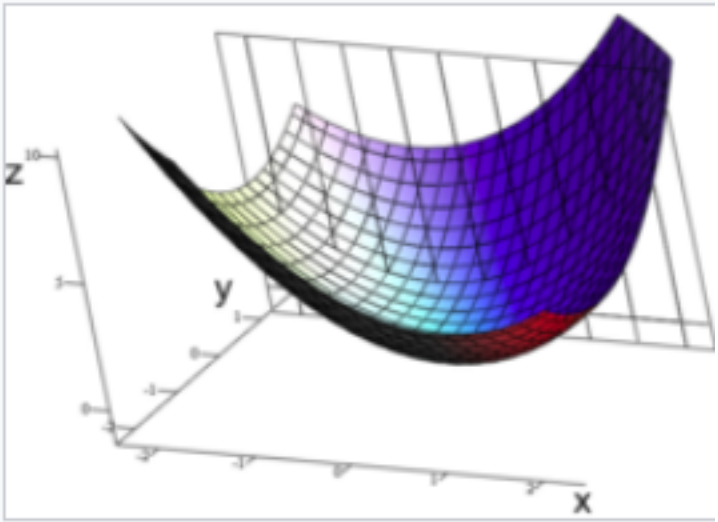
2. Adaline:

Step 1: Inner product giữa một vector đầu vào (thứ cần dc classified) và các model params

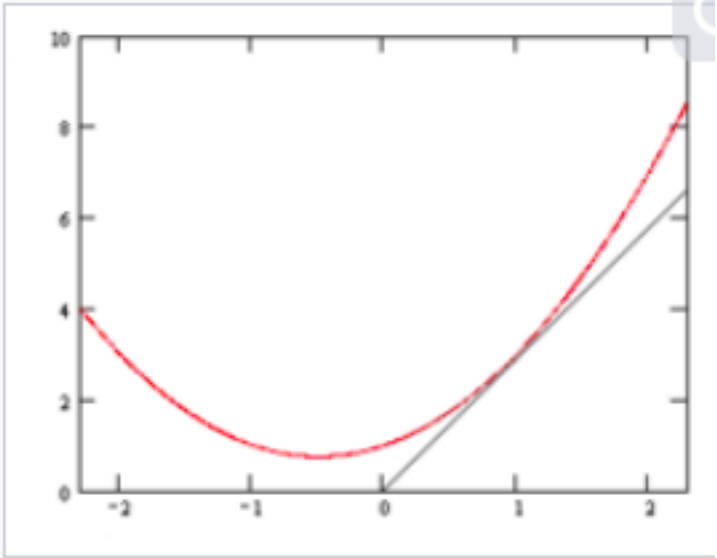
Step 2: Activation này thực ra chỉ mang tính hình thức (vì trên lý thuyết, neuron thì phải có activation func), kết quả từ Step 1 vào activation func dc giữ nguyên (như hình vẽ)

Step 3: Dùng threshold func, thường là sgn (hình vẽ cũng vẽ sgn)

1.2 Explain something about partial derivative (Đạo hàm riêng)



A graph of $z = x^2 + xy + y^2$. For the partial derivative at (1, 1, 3) that leaves y constant, the corresponding tangent line is parallel to the xz-plane.



A slice of the graph above at $y = 1$

Ta tưởng tượng về ko gian 3 chiều với hệ tọa độ Oxyz. Có một equation dc vẽ lên như hình (màu lam)

$z = x^2 + xy + y^2$

Lúc này ta có 2 biến độc lập (x, y) và 1 biến phụ thuộc (z). Đạo hàm riêng là khi bạn chỉ để 1 biến độc lập dc thay đổi giá trị, giữ cố định các biến độc lập khác. Lúc này biến phụ thuộc sẽ thay

đổi khi biến độc lập thay đổi

Giả sử ta giữ y = 1, lúc này z phụ thuộc vào sự thay đổi của x. Ta khảo sát sự biến thiên của hàm y = f(x), với hình thứ 2 ở dưới, có thể tìm giá trị đạo hàm tại 1 điểm nào đó (gradient). Thực tế ta có rất nhiều y để chọn, khi gộp hết tất cả chúng lại thì dc cái gọi là đạo hàm riêng của f theo hướng của x.

Bởi vì chúng ta giữ y, cho x chạy thì dc đạo hàm riêng f theo x cũng như tính dc giá trị đạo hàm riêng đó tại điểm nào đó trên đồ thị. Thế nên, ta cũng có thể giữ x, chạy y và làm tương tự => Làm như vậy để tại 1 điểm bất kì trên đồ thị, ta tìm dc các giá trị đạo hàm riêng theo toàn bộ các chiều của các biến độc lập, đó là vector gradient

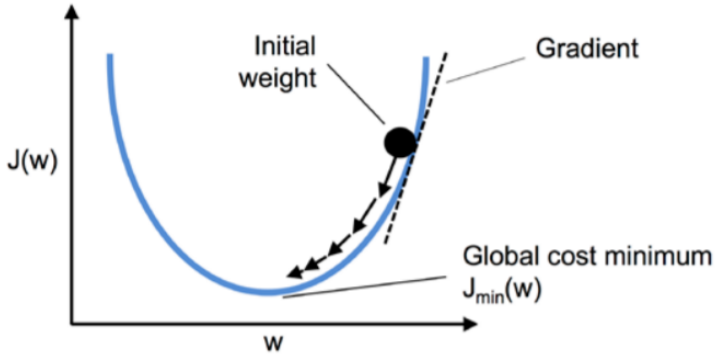
Quay lại hình đồ thị màu lam, ý nghĩa của vector gradient là: tại 1 điểm nào đó, đặt 1 hòn bi, ta xác định dc "xu hướng tiếp theo" của điểm đó trên tất cả các chiều của các biến độc lập. Một hình dung là nếu vẽ ra các tiếp tuyến tại điểm này, mỗi tiếp tuyến có độ dài và hướng theo giá trị đạo hàm riêng tương ứng, coi các tiếp tuyến là vector, cộng tất cả chúng lại, bạn sẽ thấy "hướng đi khả dĩ" của hòn bi, cứ như có trọng lực tác động vậy (mặc dù đây chỉ là một hình dung để dễ hiểu hơn cho Gradient descent, nhưng nó khá giống với inspiration về trọng lực của các phương pháp biến thể như momentum...)

2 Giảm cost function với gradient descent

Một trong những thành phần chính của thuật toán học máy có giám sát là xác định objective function và tối ưu hóa trong suốt quá trình học. Objective function thường là 1 cost function mà chúng ta muốn giảm. Với Adaline chúng ta cần định nghĩa 1 cost function J, để học weights như **sum of squared errors (SSE)** giữa kết quả tính toán và kết quả thực sự nhận được.

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

Một tính chất tuyệt vời khác của cost function này là nó lồi. Do đó ta có thể sử dụng 1 thuật toán tối ưu hóa đơn giản gọi là **gradient descent** để tìm weights và giảm cost function để phân loại. Với hình mình họa bên dưới ta có thể thấy được ý tưởng chính của **gradient descent** là ***climbing down a hill*** cho đến khi đạt được giá trị nhỏ nhất. Trong mỗi lần lặp chúng ta thực hiện 1 bước ngược hướng với gradient, ở đây kích thước của mỗi bước được xác định bởi giá trị của learning rate, cũng như độ dốc của gradient.



Dùng gradient descent ta có thể update weights bằng cách thực hiện 1 bước ngược hướng với gradient $\nabla J(w)$, của cost function $J(w)$.

$$w := w + \Delta w$$

Weiht thay đổi Δw thì được định nghĩa là âm gradient $-\nabla J(w)$ nhân với learning rate η.

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

Tính gradient của cost function chúng ta cần tính đạo hàm từng phần của cost function đối với mỗi weights w_j.

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

Vì vậy có thể viết công thức cập nhật của w_j là:

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

Hơn nữa cập nhật weights tính toán dựa trên tất cả training dataset (thay vì cập nhật giảm weights sau mỗi lần training mẫu) đó là lý do tại sao phương pháp này cũng được gọi là **batch gradient descent**.

3 Thực hiện Adaline trên python

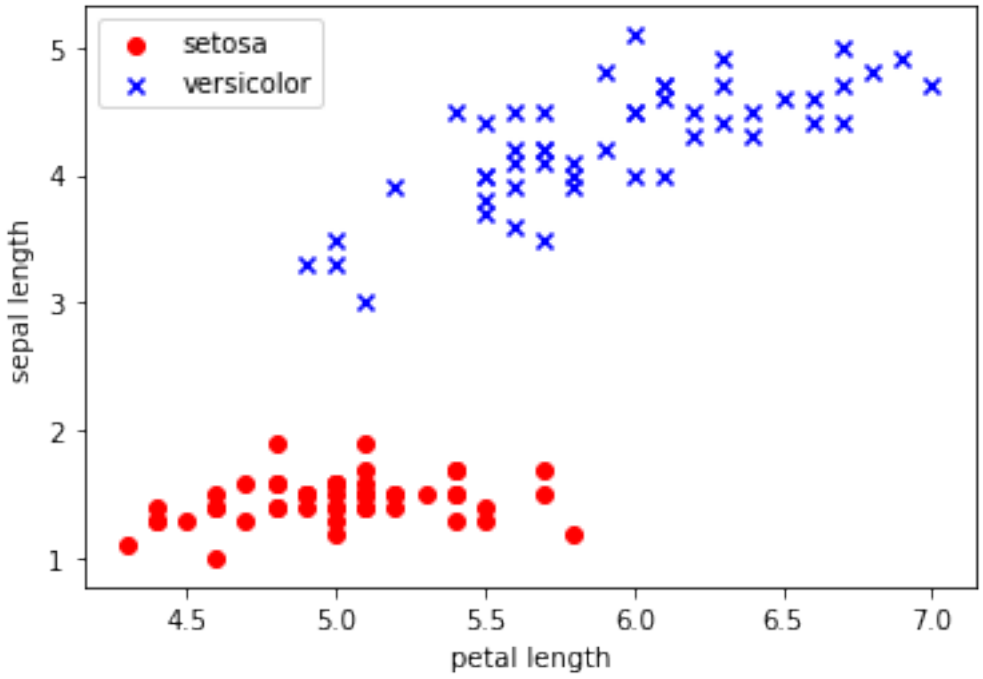
3.1 Source code

[36]:
```
import os
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
s = os.path.join('https://archive.ics.uci.edu', 'ml',␣
 ↪'machine-learning-databases', 'iris','iris.data')
df = pd.read_csv(s.replace("\\", "/"), header=None, encoding='utf-8')
```

[37]:
```
y = df.iloc[0:100, 4].values
y = np.where(y == 'Iris-setosa', -1, 1)
X = df.iloc[0:100, [0, 2]].values
```

Visualization cho dataset

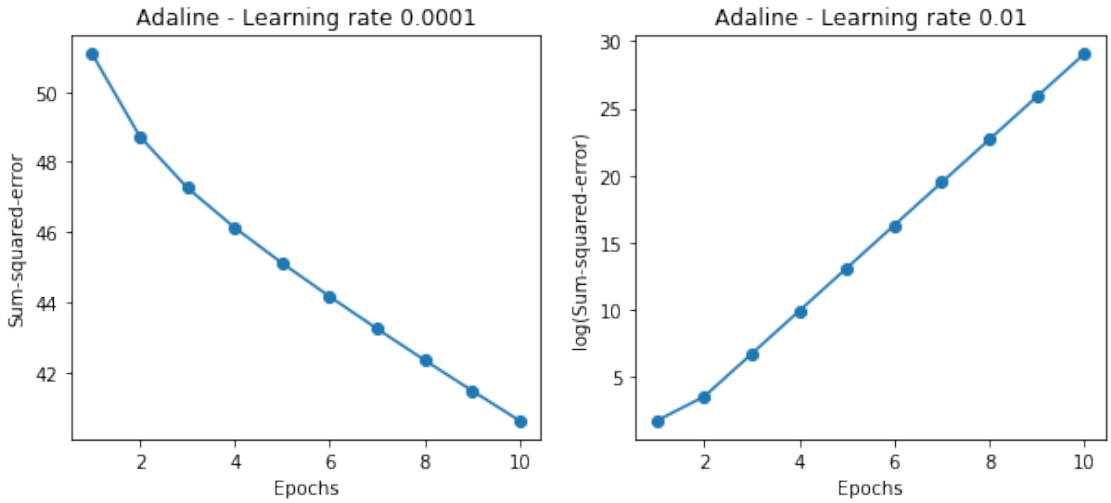
[38]:
```
plt.scatter(X[:50, 0], X[:50, 1], color = 'red', marker = 'o', label = 'setosa')
plt.scatter(X[50:100, 0], X[50:100, 1], color = 'blue', marker = 'x', label =␣
 ↪'versicolor')
plt.xlabel('petal length')
plt.ylabel('sepal length')
plt.legend(loc = 'upper left')
plt.show()
```

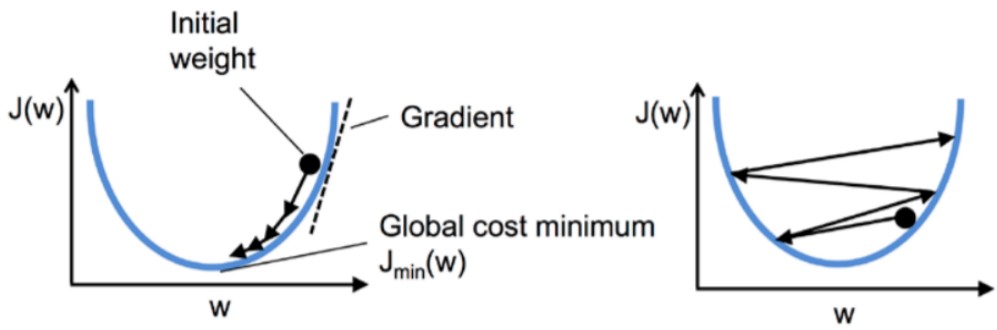


[39]:
```
class AdalineGD(object):
    def __init__(self, eta=0.01, n_iter=50, random_state=1):
        self.eta = eta
        self.n_iter = n_iter
        self.random_state = random_state
    def fit(self, X, y):
        rgen = np.random.RandomState(self.random_state)
        self.w_ = rgen.normal(loc=0.0, scale=0.01,
        size=1 + X.shape[1])
        self.cost_ = []

        for i in range(self.n_iter):
            net_input = self.net_input(X) # dot prod X (100,2) @ w (2,1) =>␣
↪(100,1)
            output = self.activation(net_input) # output (100,1)
            errors = (y - output) # (100,1)
            self.w_[1:] += self.eta * X.T.dot(errors) # (2,100) @ (100,1) =>␣
↪(2,1) grad vec
            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)
```

[40]:
```
from sklearn import datasets

fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(10, 4))
ada1 = AdalineGD(n_iter=10, eta=0.01).fit(X, y)

ax[1].plot(range(1, len(ada1.cost_) + 1), np.log10(ada1.cost_), marker='o')
ax[1].set_xlabel('Epochs')
ax[1].set_ylabel('log(Sum-squared-error)')
ax[1].set_title('Adaline - Learning rate 0.01')

ada2 = AdalineGD(n_iter=10, eta=0.0001).fit(X, y)
ax[0].plot(range(1, len(ada2.cost_) + 1),ada2.cost_, marker='o')
ax[0].set_xlabel('Epochs')
ax[0].set_ylabel('Sum-squared-error')
ax[0].set_title('Adaline - Learning rate 0.0001')
plt.show()
```





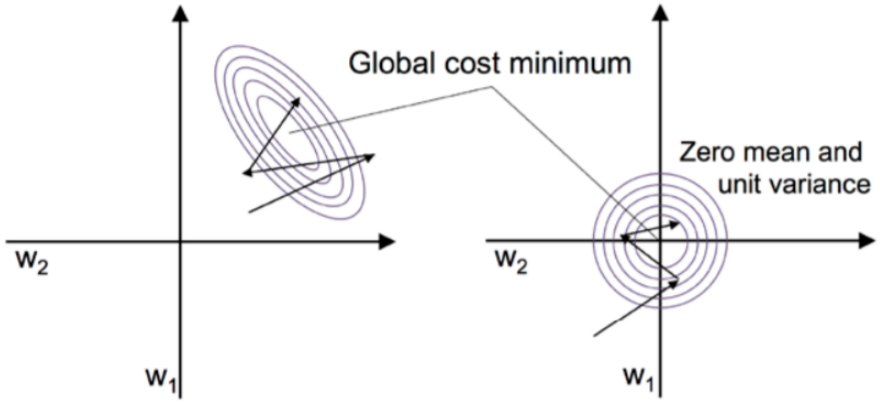
Như chúng ta cũng có thể thấy kết quả của biểu đồ cost function, chúng ta có 2 vấn đề sau. Biểu đồ bên phải cho ta thấy những gì xảy ra khi ta chọn learning rate quá lớn. Thay vì giảm cost function, error trở nên lớn hơn sau mỗi epoch, bởi vì chúng đã ***overshoot*** mức global minimum. Ngược lại chúng ta có thể thấy cost giảm ở biểu đồ bên trái. Tuy nhiên learning rate $\eta = 0.0001$ thì quá nhỏ mà điều này yêu cầu cần phải có số lượng lớn epoch để có thể hội tụ về global minimum.

3.2 Cải thiện gradient descent thông qua feature scaling

Chúng ta sẽ dùng phương thức feature scaling được gọi là **standardization**. Cung cấp cho dữ liệu của chúng ta tính chất của phân phối chuẩn: **zero-mean** và **unit variance**. Thủ tục chuẩn hóa này giúp cho gradient descent hội tụ nhanh hơn. Ví dụ: cho standardize của feature jth, chúng ta có thể trừ trung bình của mẫu μ_j và chia nó với độ lệch chuẩn σ_j.

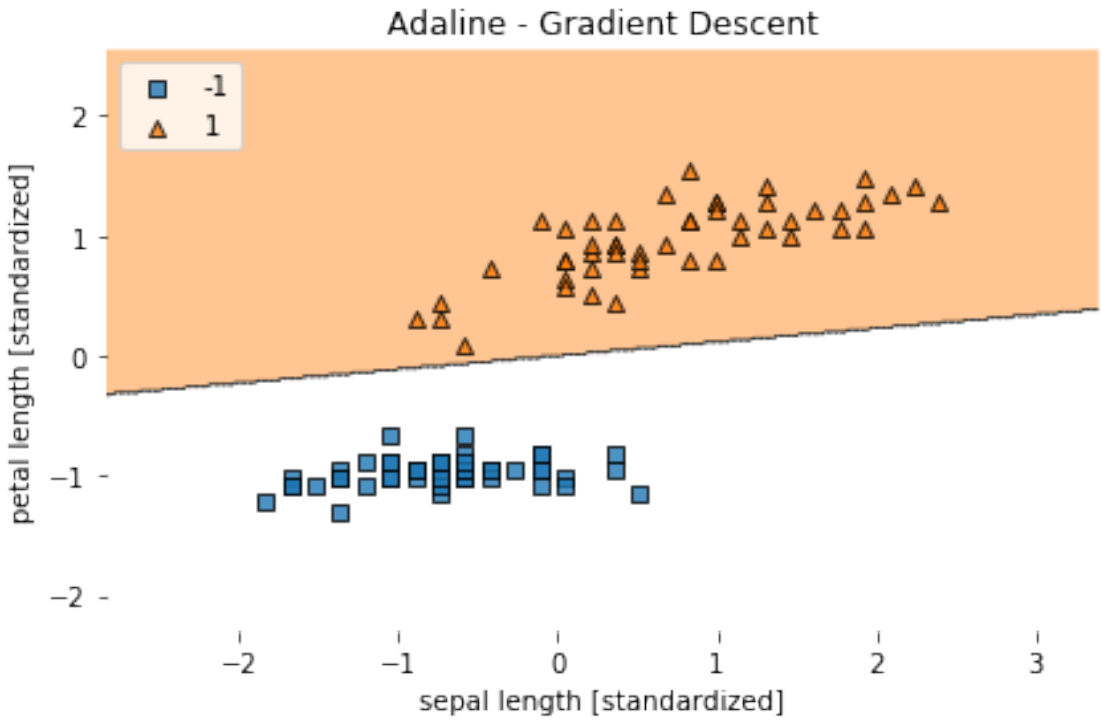
$$x_j^{'} = \frac{x_j - \mu_j}{\sigma_j}$$

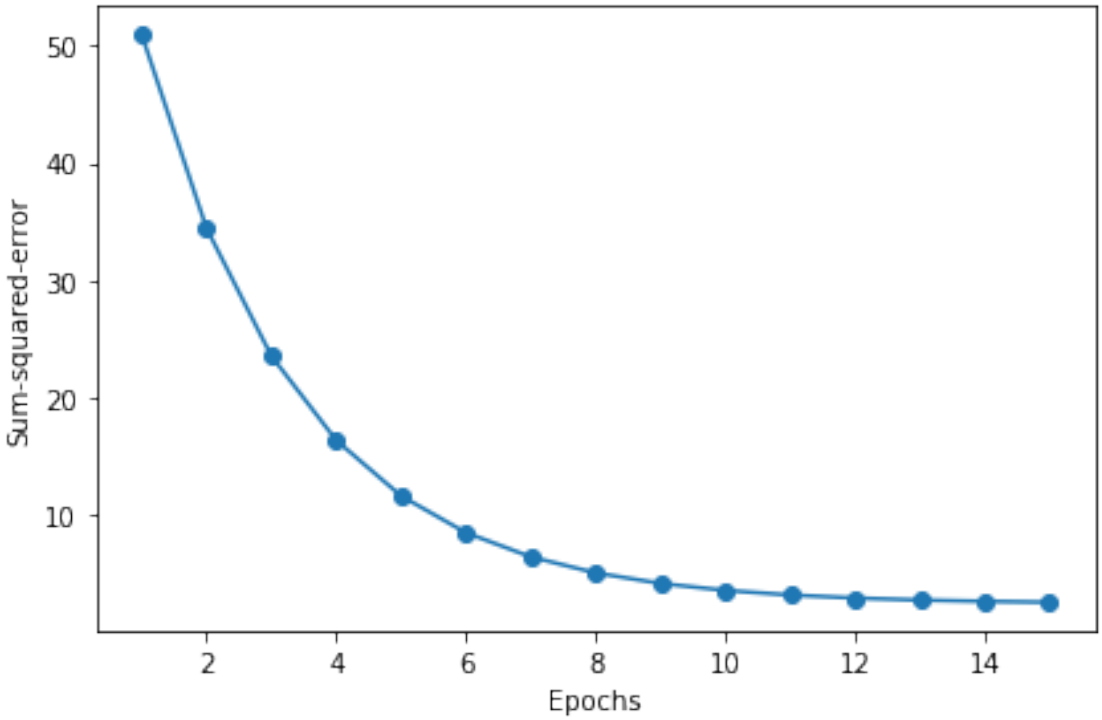
Một trong những lý do tại sao standardization giúp cho gradient descent tối ưu hóa hơn là trải qua ít bước hơn để tìm ra giải pháp tối ưu (global cost minimum). Nó đi vuông góc với các đường đồng mức! Như ta có minh họa sau:



[41]:
```
from mlxtend.plotting import plot_decision_regions

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_gd = AdalineGD(n_iter=15, eta=0.01)
ada_gd.fit(X_std, y)

plot_decision_regions(X_std, y, clf=ada_gd)
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_gd.cost_) + 1), ada_gd.cost_, marker='o')
plt.xlabel('Epochs')
plt.ylabel('Sum-squared-error')
plt.tight_layout()
plt.show()
```





4 Học máy quy mô lớn và stochastic gradient descent, mini batch gradient descent

4.1 Stochastic gradient descent

Trong phần trước, chúng ta đã học làm thế nào để giảm cost function bởi các bước ngược hướng với cost gradien từ toàn bộ training dataset, đó là lý do tại sao cách tiếp cận này còn được gọi là batch gradient descent. Bây giờ hãy tưởng tượng nếu chúng ta có bộ dataset có hàng triệu training example, chạy batch gradient descent rất tốn kém trong các tình huống như thế vì chúng ta cần đánh giá lại toàn bộ tập dữ liệu mỗi lần chúng ta thực hiện 1 bước để di chuyển đến global gradient minimum. Thay thế phổ biến của batch gradient descent là **stochastic gradient descent (SGD)** ngoài ra còn có thể gọi cách cách khác là iterative hoặc online gradient descent. Thay vì update weights dựa trên tổng lỗi tích lũy trên tất cả training sample x^i.

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

Chúng ta cập nhật weights giảm cho mỗi training example.

$$\eta(y^{(i)} - \phi(z^{(i)}))x^{(i)}$$

Mặc dù SGD xem như xấp xỉ gradient descent, nó hội tụ nhanh hơn vì thường xuyên cập nhật weights, mỗi gradient thì được tính toán dựa trên 1 training example. Điều này có lợi thế là SGD

có thể thoát khỏi local minimum nhanh hơn nếu chúng ta làm việc với nonlinear cost function. Và một điều nữa, cần xáo trộn dataset cho mỗi epoch để ngăn algo (vốn xử lý tuần tự trên lần lượt các training example) học một cách lặp lại, tránh algo hiểu ngầm rằng các training example theo 1 thứ tự nào đó có liên kết với nhau. **Điều chỉnh learning rate trong suốt quá trình training:** Trong quá trình thực hiện SGD, fixed learning rate thường được thay thế bởi adaptive learning rate giảm dần theo thời gian. Ví dụ:

$$\frac{c_1}{[numberofiterations] + c_2}$$

Ở đây c_1 và c_2 là 1 hằng số. **Chú ý:** SGD không đạt global minimum mà chỉ rất gần với nó. Và dùng adaptive learning rating chúng ta có thể đạt được cost minimum.

Cải tiến khác của SGD là chúng ta có thể dùng nó cho online learning rate. Trong online learning, model của chúng ta có thể traning ngay khi dữ liệu mới đến. Điều này đặc biệt hữu ích khi ta có lượng lớn dữ liệu tích lũy. Dùng online learning, hệ thống có thể thích nghi ngay với những thay đổi và dữ liệu training có thể được bỏ đi sau khi cập nhật model nếu kho lưu trữ có hạn.

4.2 Mini-batch gradient descent

Sự kết hợp giữa batch gradient descent và SGD còn được gọi là mini-batch gradient descent. Mini-batch gradient descent có thể được hiểu là SGD cho các tập dữ liệu con có kích thước nhất định (mini-batch). Nó có lợi thế hơn batch gradient descent là hội tụ nhanh hơn vì weights được cập nhật thường xuyên. Hơn thế nữa mini-batch gradient descent cho phép chúng ta thay thế vòng lặp for bằng cách tận dụng khái niệm vector hóa từ đại số tuyến tính.

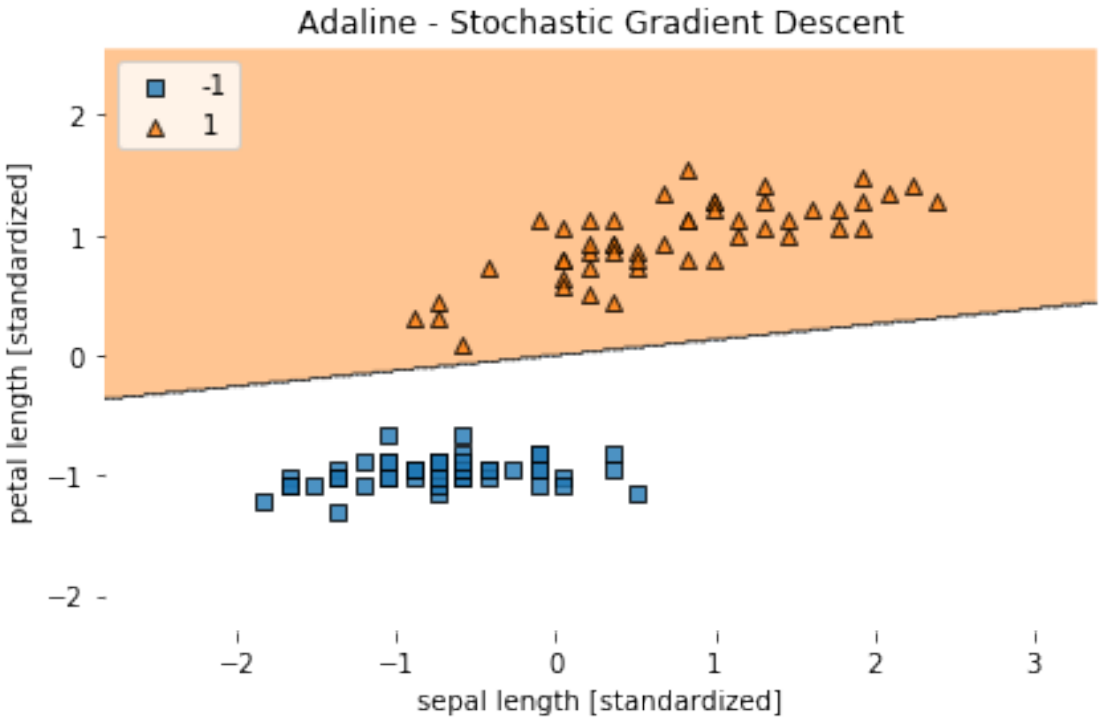
Hiện thực hóa thuật toán Adaline SGD với python:

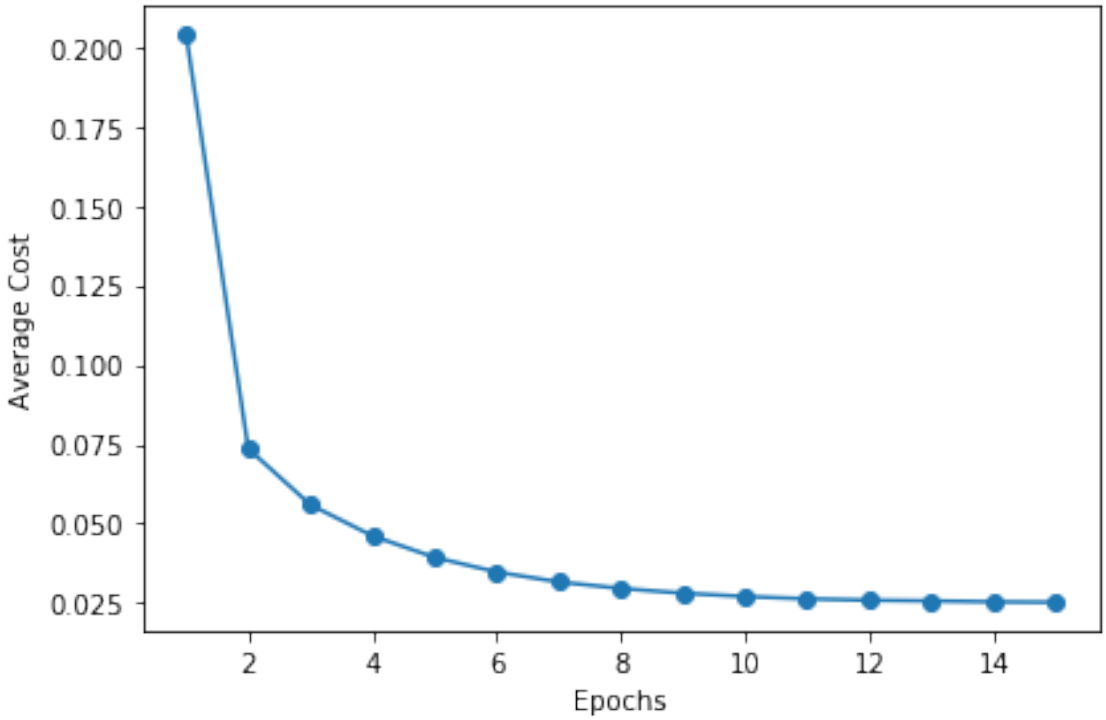
[42]:
```
class AdalineSGD(object):
    def __init__(self, eta=0.01, n_iter=10,
        shuffle=True, random_state=None):
        self.eta = eta
        self.n_iter = n_iter
        self.w_initialized = False
        self.shuffle = shuffle
        self.random_state = random_state

    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)
```

[43]:
```
ada_sgd = AdalineSGD(n_iter=15, eta=0.01, random_state=1)
ada_sgd.fit(X_std, y)
plot_decision_regions(X_std, y, clf=ada_sgd)
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_sgd.cost_) + 1), ada_sgd.cost_, marker='o')
plt.xlabel('Epochs')
plt.ylabel('Average Cost')
plt.tight_layout()
plt.show()
```





Như các bạn có thể thấy chi phí trung bình giảm khá nhanh và hoàn thành sau 15 epoch trông tương tự như gradient descent Adaline. Nếu chúng ta muốn cập nhật model trên online learning với streaming data. Chúng ta có thể gọi phương thức partial_fit. Ví dụ: ada_sgd.partial_fit(X_std[0, :], y[0]).