perceptron_basics

July 5, 2018

1 Training Machine Learning Algorithms for Classification

Python exercises on training machine learning algorithms for classification. This notebook is based on Chapter 2 of the book Python Machine Learning by Sebastian Raschka.

- Code Repository: https://github.com/rasbt/python-machine-learning-book-2nd-edition
- Notebook Reference: `https://github.com/rasbt/python-machine-learning-book-2nd-edition/blob/master/code/ch02/ch02.ipynb`

- Section 1.1
  - Section 1.1.1
  - Section 1.1.2
- Section 1.2
  - Section 1.2.1
  - Section 1.2.2
  - Section 1.2.3
- Section 1.3
  - Section 1.3.1
  - Section 1.3.2
  - Section 1.3.3
- Section 1.4

In this notebook, we will be implementing two algorithms of classification applied to linearly seperable data:

- Rosenblatt, Frank. "The perceptron: a probabilistic model for information storage and organization in the brain." Psychological review 65.6 (1958): 386.
- An Adaptive "Adaline" Neuron Using Chemical "Memistors", Technical Report Number 1553-2, B. Widrow and others, Stanford Electron Labs, Stanford, CA, October 1960

```
In [1]: # import required libraries

        # array manipulation and linear algebra library
        import numpy as np

        # library to handle raw data files
        import pandas as pd

        # plotting library
        import matplotlib.pyplot as plt

        # to load images into the notebook
        from IPython.display import Image

        # for drawing the decision regions
        from matplotlib.colors import ListedColormap

        %matplotlib inline
```

1.1 Data Processing

1.1.1 Iris Dataset

We will use the IRIS dataset to conduct our experiments. This can be downloaded from the UCI machine learning repository.

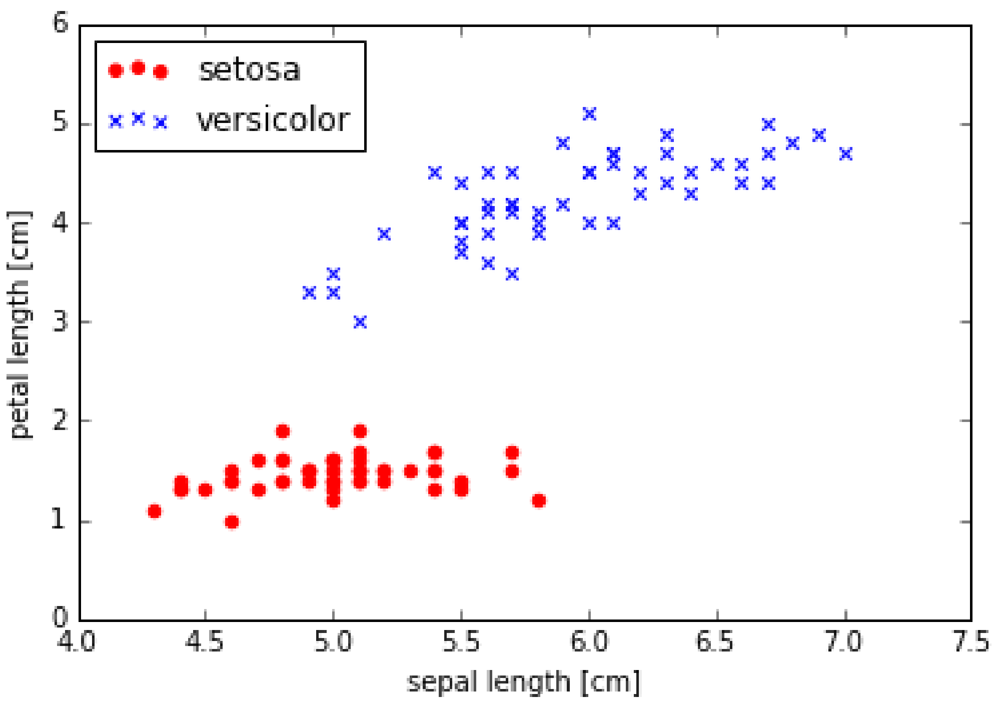
```
In [2]: ## loading the iris dataset

        # use pandas library to access url
        df = pd.read_csv('https://archive.ics.uci.edu/ml/'
                'machine-learning-databases/iris/iris.data', header=None)

        # print the last few samples of iris dataset
        df.tail()
```

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

```
In [3]: ## plotting the iris dataset

        # 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

# 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')

# add x-label, y-label and legend to the plot
plt.xlabel('sepal length [cm]')
plt.ylabel('petal length [cm]')
plt.legend(loc='upper left')

# display the figure
plt.show()
```



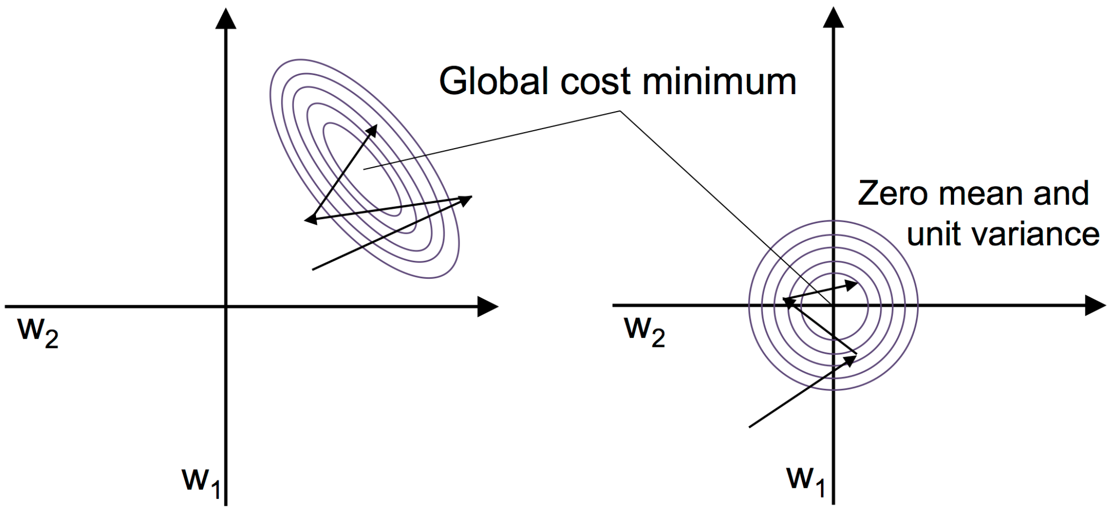
1.1.2 Feature Scaling

Feauture scaling is a very important step to ensure that training the model becomes stable. The most commonly used approach is the standard normalization which is implemented as follows:

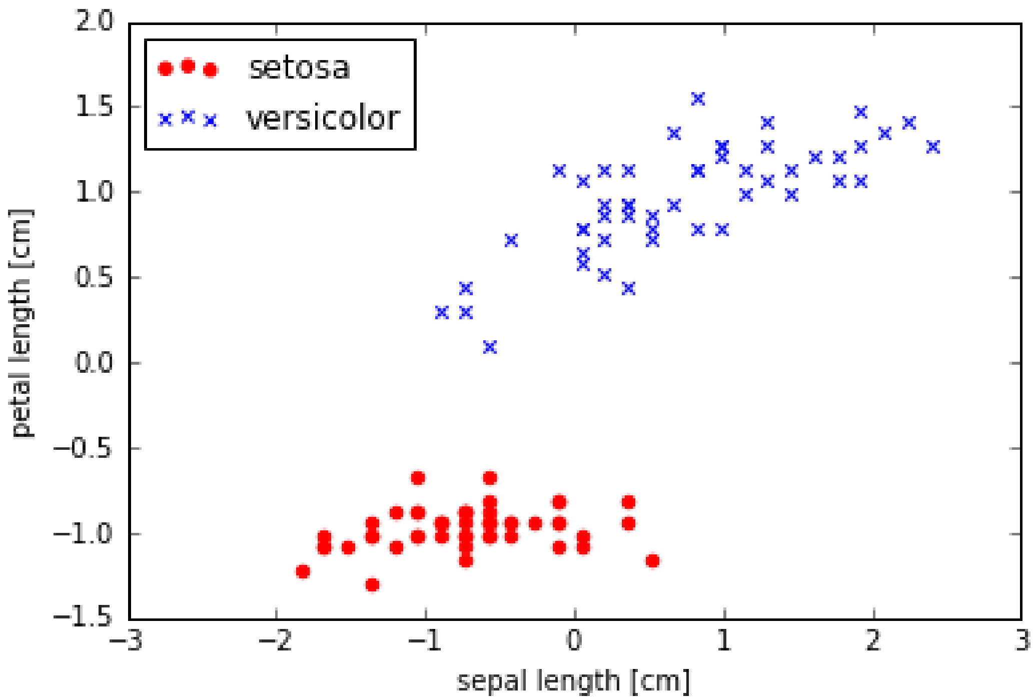
$$x'_j = \frac{x_j - \mu_j}{\sigma_j}$$

```
In [4]: Image(filename='./images/img6.png', width=700)
```

Out[4]:



In [6]:
```
# create a copy of the X data matrix
X_std = np.zeros(X.shape)

# implement the standard normalization to the variable X_std
########################## BEGIN CODE ########################

########################### END CODE ###########################

# plot the standardized data
plt.scatter(X_std[:50, 0], X_std[:50, 1],
            color='red', marker='o', label='setosa')
plt.scatter(X_std[50:100, 0], X_std[50:100, 1],
            color='blue', marker='x', label='versicolor')

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

# display the figure
plt.show()
```



1.2 Perceptron Learning Algorithm

1.2.1 Perceptron Learning Rule

The perceptron algorithm uses a unit step function as the decision function $\phi(\cdot)$:

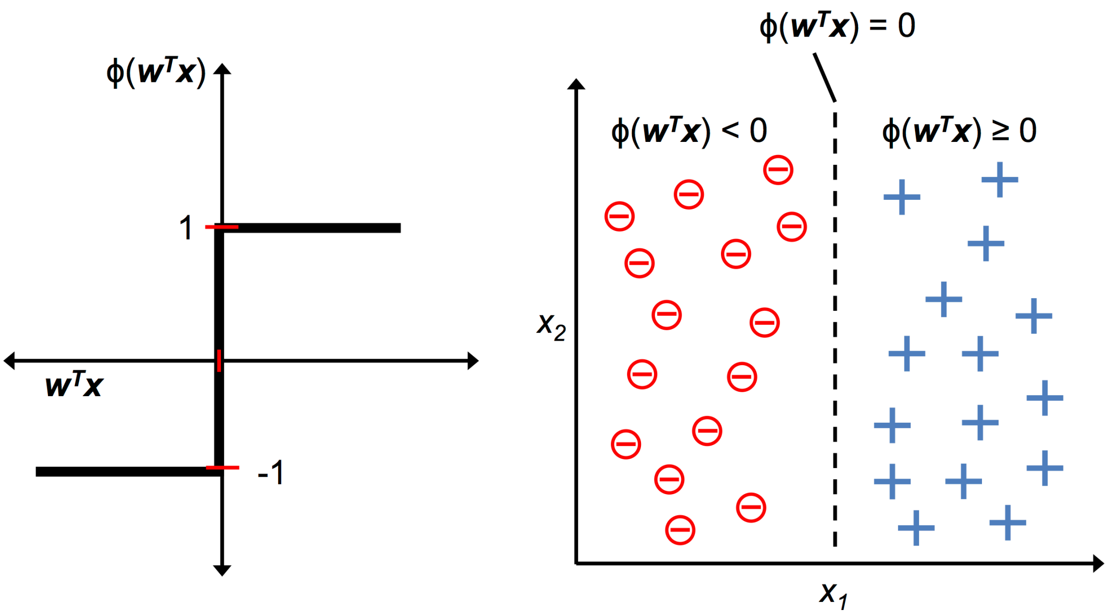
$$\phi(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ -1 & \text{otherwise} \end{cases}$$

where z is given by:

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

```
In [7]: Image(filename='./images/img1.png', width=500)
```

Out[7]:



The update rule for the perceptron is given by:

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

where the update coefficient Δw_j is given by:

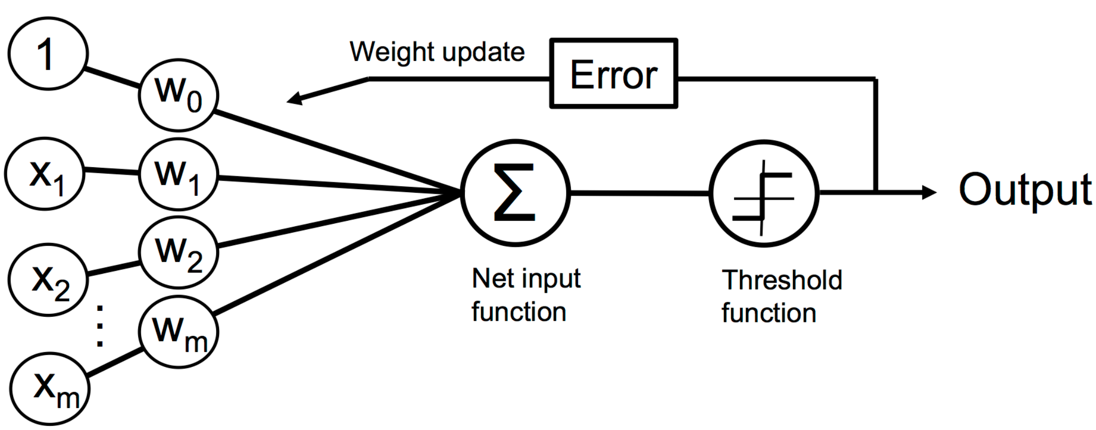
$$\Delta w_j = \eta \sum_i (y^{(i)} - \hat{y}^{(i)}) x_j^{(i)}$$

this will be represented by a set of equations as follows:

$$\Delta w_0 = \eta \sum_i (y^{(i)} - \hat{y}^{(i)}) \Delta w_1 = \eta \sum_i (y^{(i)} - \hat{y}^{(i)}) x_1^{(i)} \Delta w_2 = \eta \sum_i (y^{(i)} - \hat{y}^{(i)}) x_2^{(i)}$$

In [8]:
```
Image(filename='./images/img2.png', width=600)
```

Out[8]:



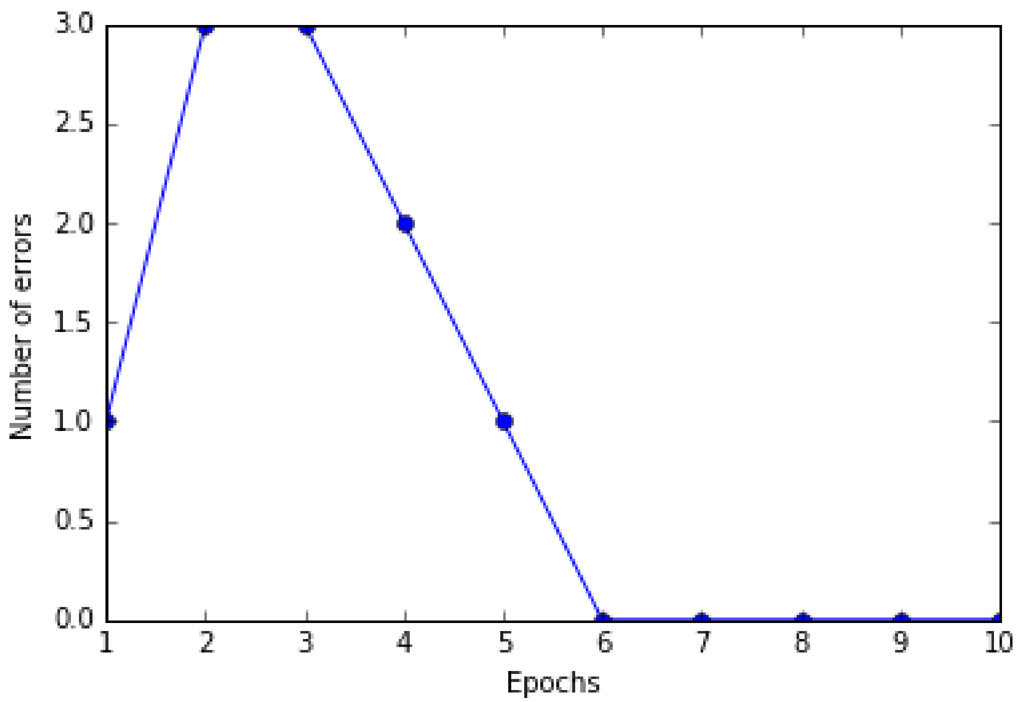
1.2.2 Object-oriented Implementation

```
In [17]: class Perceptron(object):
             """Perceptron classifier.

             Parameters
             ------------
             eta : float
               Learning rate (between 0.0 and 1.0)
             n_iter : int
               Passes over the training dataset.
             random_state : int
               Random number generator seed for random weight
               initialization.

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

             """
             def __init__(self, eta=0.01, n_iter=50, random_state=1):

                 # initialize class variables using input arguments

                 # eta represents learning rate
                 self.eta = eta

                 # n_iter represents number of iterations on training dataset
                 self.n_iter = n_iter

                 # random_state is the seed used for randomization
                 self.random_state = random_state

             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]
```

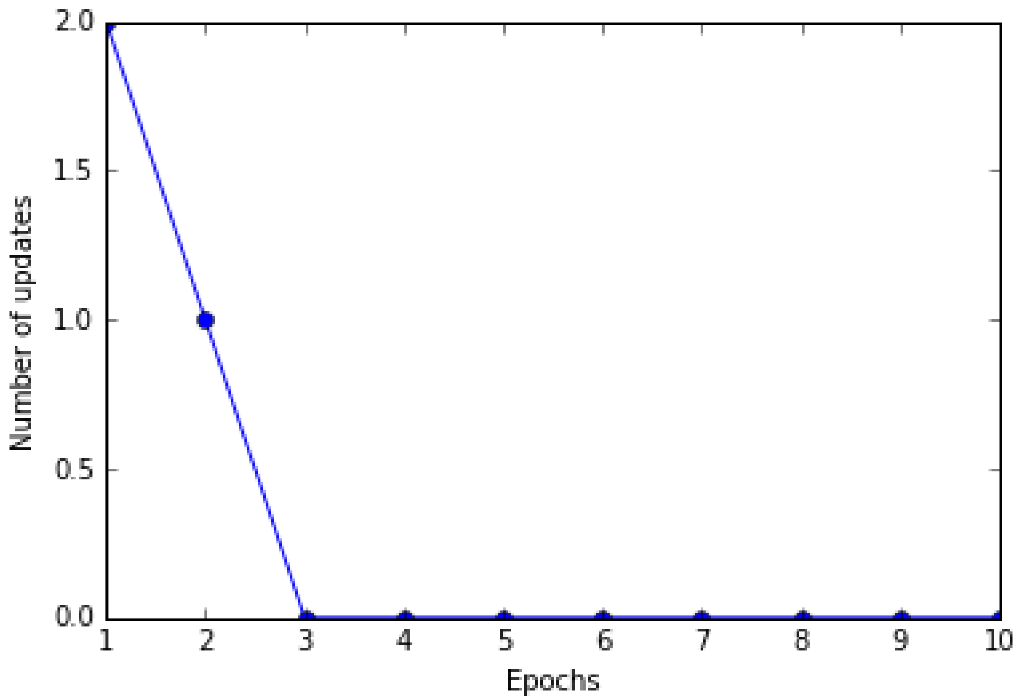
```
              Target values.

            Returns
            -------
            self : object

            """
            # initialize random initial weights of perceptron
            rgen = np.random.RandomState(self.random_state)
            self.w_ = rgen.normal(loc=0.0, scale=0.01, size=1 + X.shape[1])

            # initialize array of errors in each iteration
            self.errors_ = [0]*self.n_iter

            # setup a for loop for n_iter iterations
            for n in range(self.n_iter):
                # initialize error for
                errors = 0

                ################## BEGIN CODE ############################
                # hint: loop over all the training samples
                # for each sample, use self.predict to get activation
                # use activation to estimate the weight update Delta w
                # update the weights self.w_ using the Delta w
                # return the number of errors for keep tracking of progress

                ################### END CODE ##############################

                self.errors_[n] = errors

            return self

        def net_input(self, X):
            """Calculate net input"""
            # this function computes the output of the perceptron using weights w_
            return np.dot(X, self.w_[1:]) + self.w_[0]

        def predict(self, X):
            """Return class label after unit step"""

            # this function implements the unit step activation function phi
            return np.where(self.net_input(X) >= 0.0, 1, -1)
```

In [18]:
```
# create an instance of perceptron
ppn = Perceptron(eta=0.01, n_iter=10)

# perform training on the dataset
ppn.fit(X, y)
```

```
# visualize the training error as a function of iterations
plt.plot(range(1, len(ppn.errors_) + 1), ppn.errors_, marker='o')

# add labels
plt.xlabel('Epochs')
plt.ylabel('Number of errors')

# display the figure
plt.show()
```



In [19]:
```
# perform training on the standardized dataset
ppn.fit(X_std, y)

# visualize the training error
plt.plot(range(1, len(ppn.errors_) + 1), ppn.errors_, marker='o')

# add labels to the plot
plt.xlabel('Epochs')
plt.ylabel('Number of updates')

# display the figure
plt.show()
```

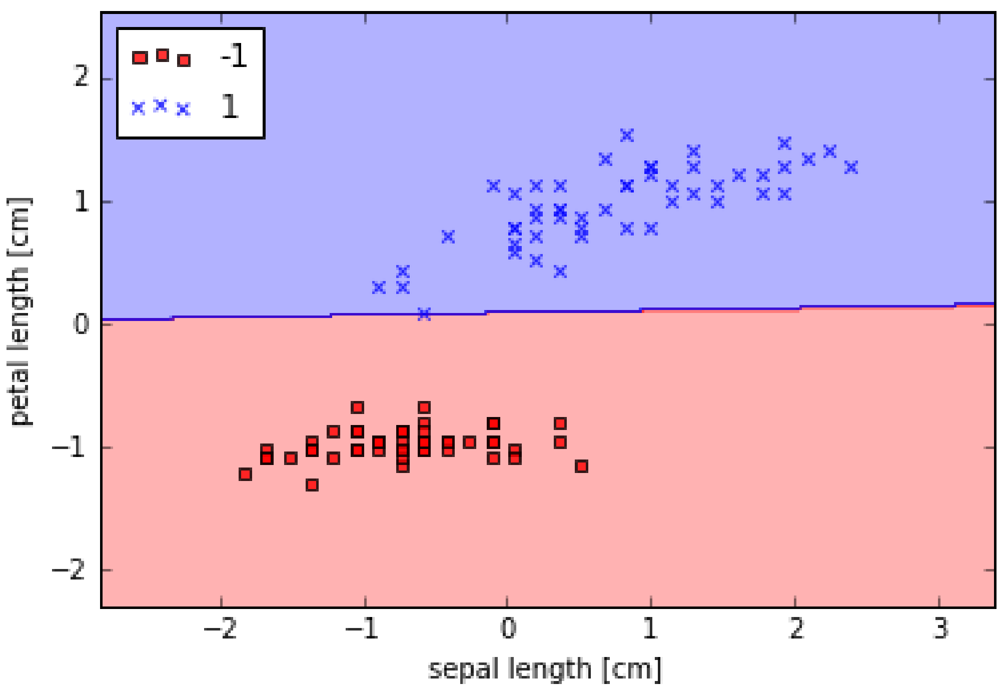


1.2.3 Plotting Decision Regions

```
In [20]: def plot_decision_regions(X, y, classifier, resolution=0.02):
             """Utility function to plot decision regions of perceptron"""

             # 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))
             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())

             # plot class samples
             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')
```

```
In [21]: # plot the decision region for the previous example
         plot_decision_regions(X_std, y, classifier=ppn)

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

         # display the figure
         plt.show()
```



1.3 Adaptive Linear Neurons

1.3.1 Minimizing Cost Functions

The Adaline algorithm is an updated version of the perceptron algorithm. Here the activation function is just the identity function:

$$\phi(\mathbf{w}^T\mathbf{x}) = \mathbf{w}^T\mathbf{x}$$

The cost function for training the algorithm is given by:

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

Based on this cost function, the weight update rule is given by:

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

where $\Delta\mathbf{w}$ is given by:

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

The gradient for each weight w_j is given by:

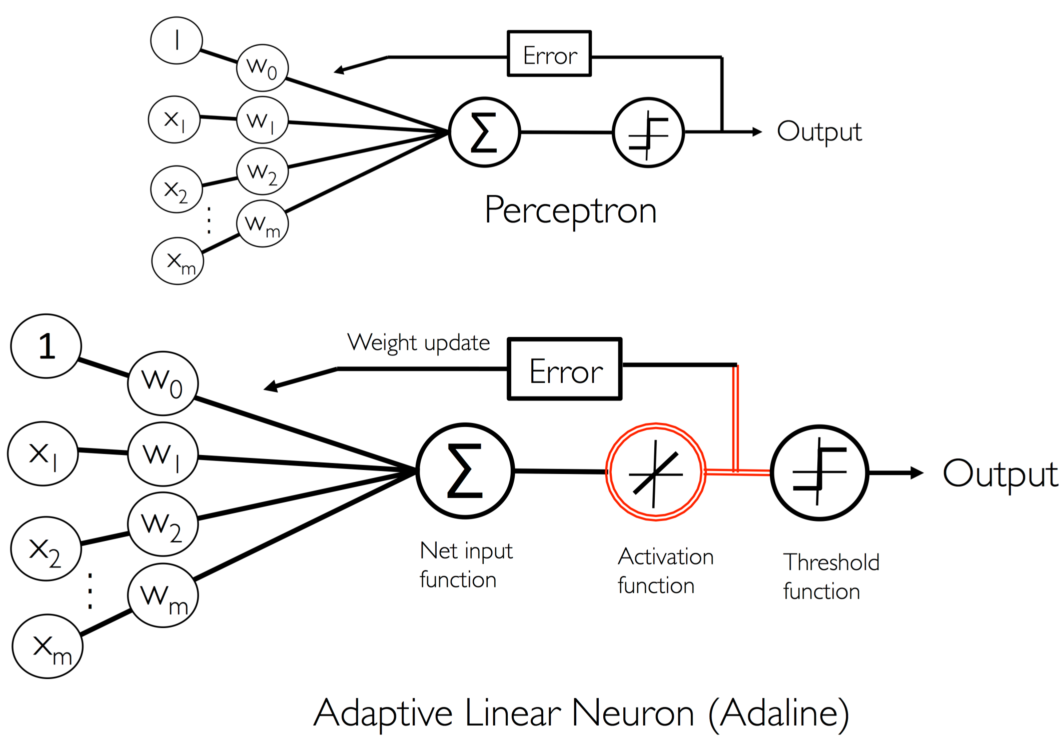
$$\frac{\partial J}{\partial w_j} = -\sum_i \left(y^{(i)} - \phi(z^{(i)})\right) x_j^{(i)}$$

This implies:

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

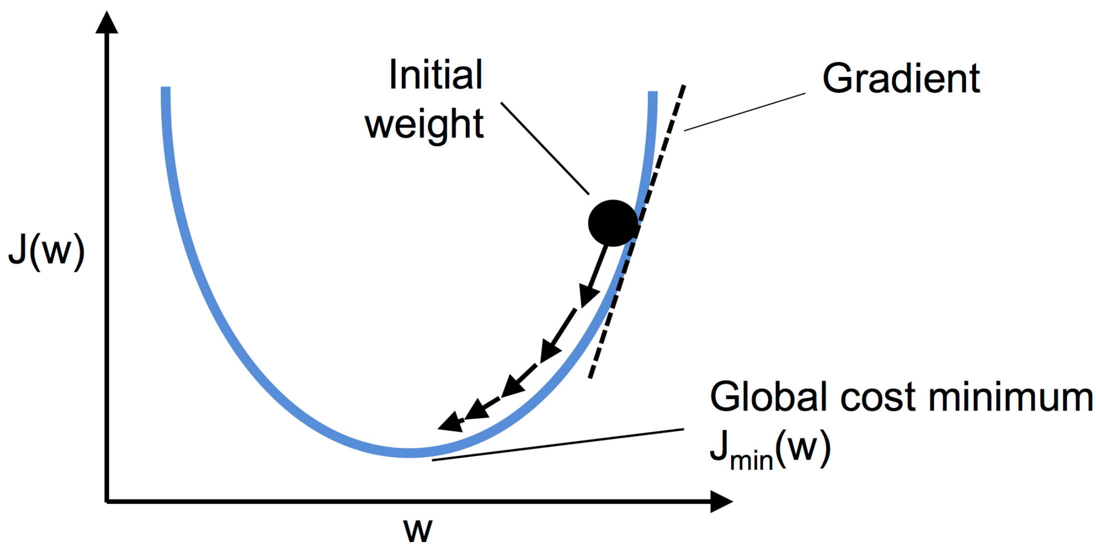
```
In [22]: Image(filename='./images/img3.png', width=600)
```

Out[22]:



```
In [23]: Image(filename='./images/img4.png', width=500)
```

Out[23]:



1.3.2 Implementing Adaptive Linear Neuron

```
In [27]: 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.
             random_state : int
               Random number generator seed for random weight
               initialization.


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

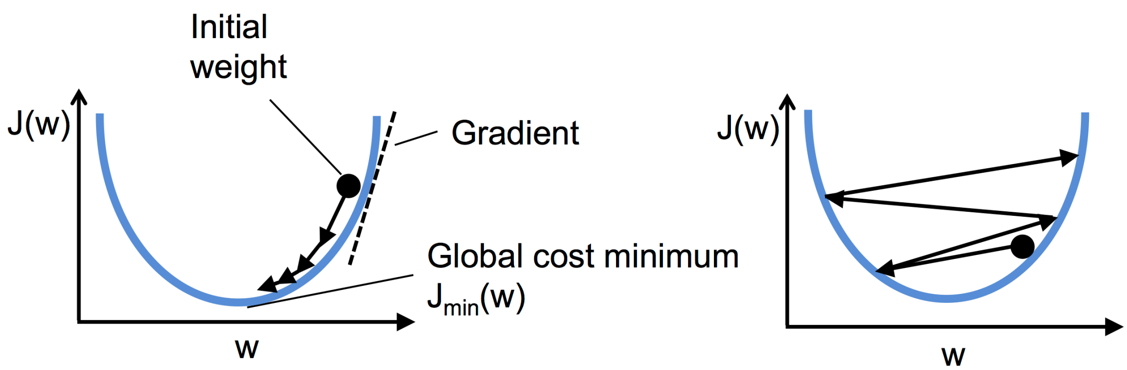
```
    """
    def __init__(self, eta=0.01, n_iter=50, random_state=1):

        # initialize class variables using input arguments
        self.eta = eta
        self.n_iter = n_iter
        self.random_state = random_state

    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

        """
        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):
            cost = 0

            #################### BEGIN CODE #################################
            # hint: obtain output of perceptron using the functions
            # self.net_input and self.activation
            # compute the errors using the output and obtain the Delta w
            # compute the overall cosr J(w) to keep track of performance

            #################### END CODE ###################################

            self.cost_.append(cost)
        return self

    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 X

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

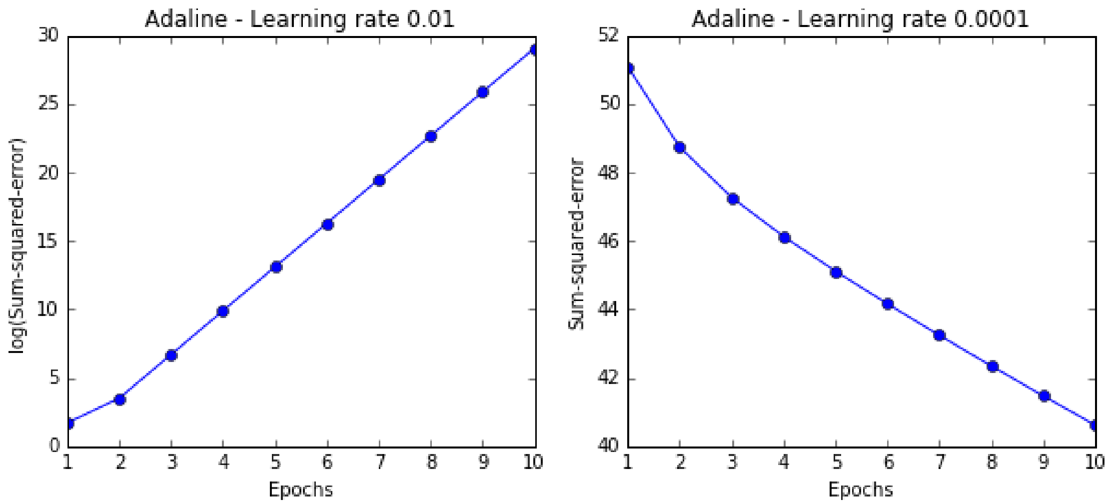
1.3.3 Effect of Learning Rate

In [28]:
```
Image(filename='./images/img5.png', width=700)
```

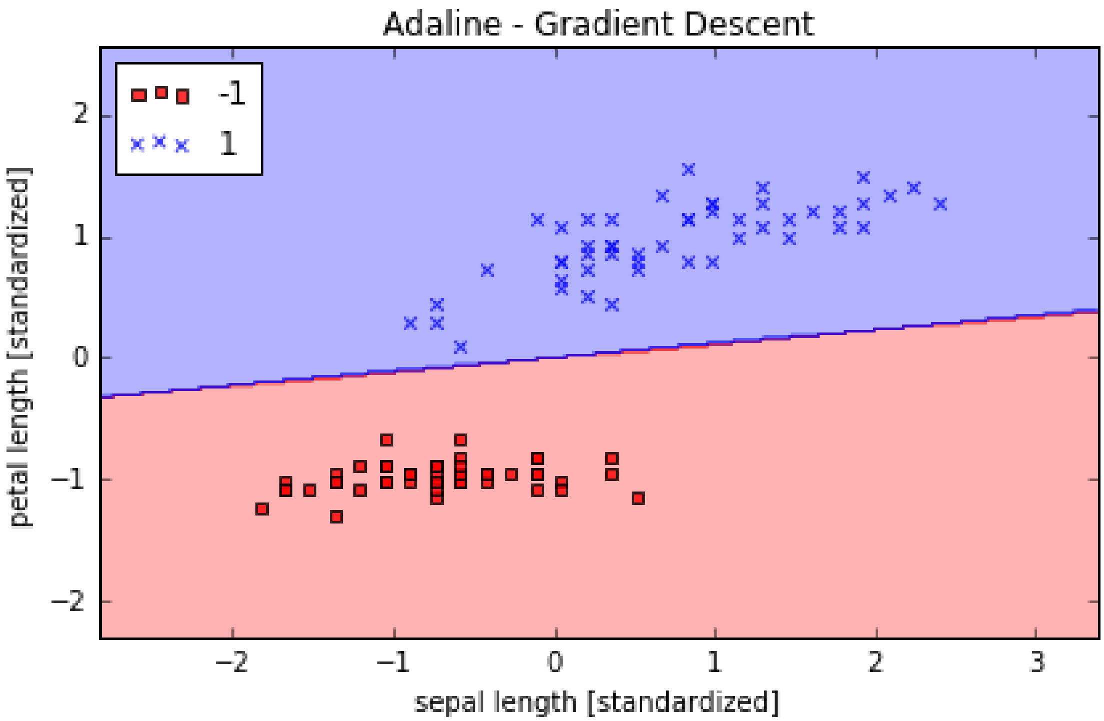
Out[28]:

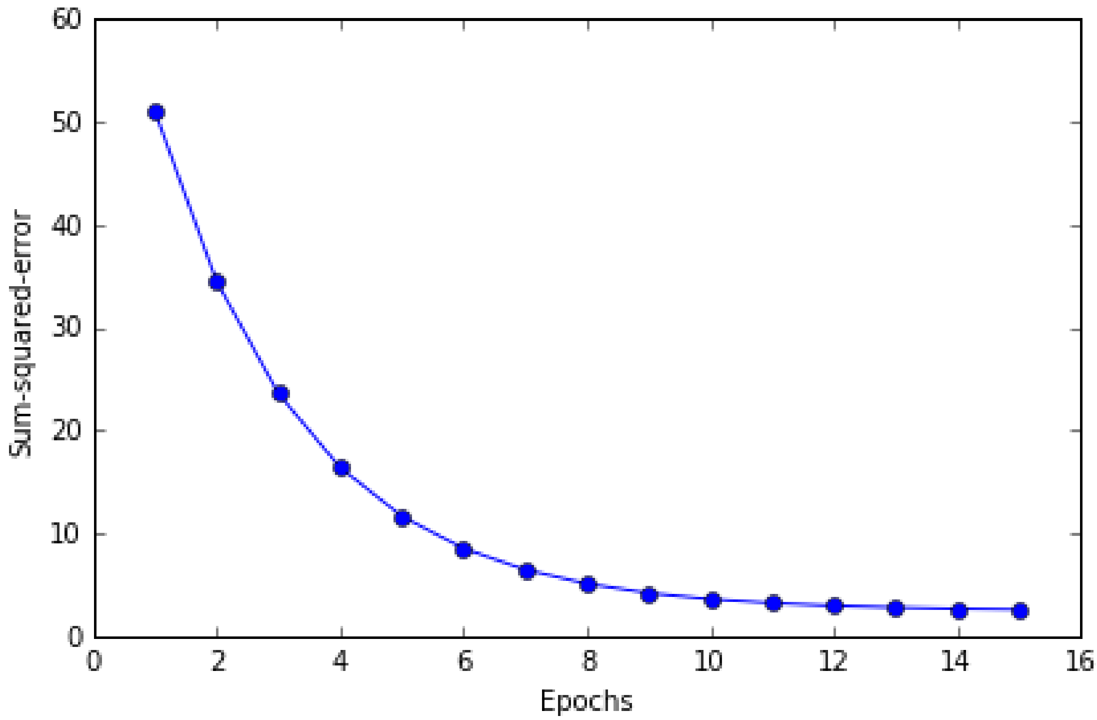


In [29]:
```
# show the effect of learning rate on performance of the algorithm
fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(10, 4))

# instance 1 has learning rate of 0.01
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')

# instance 2 has learning rate of 0.0001
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.savefig('images/02_11.png', dpi=300)
plt.show()
```



In [30]:
```
# apply the adaline algorithm to standardized dataset
ada = AdalineGD(n_iter=15, eta=0.01)
ada.fit(X_std, y)

# plot decision regions of the algorithm
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()

# plot the learning curve of the algorithm
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()
```


Adaline - Gradient Descent
-1
1
petal length [standardized]
sepal length [standardized]


Sum-squared-error
Epochs

1.4 Stochastic Gradient Descent

Typically gradient descent is applied in a batch on the whole training dataset. This can be replaced as follows:

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

with

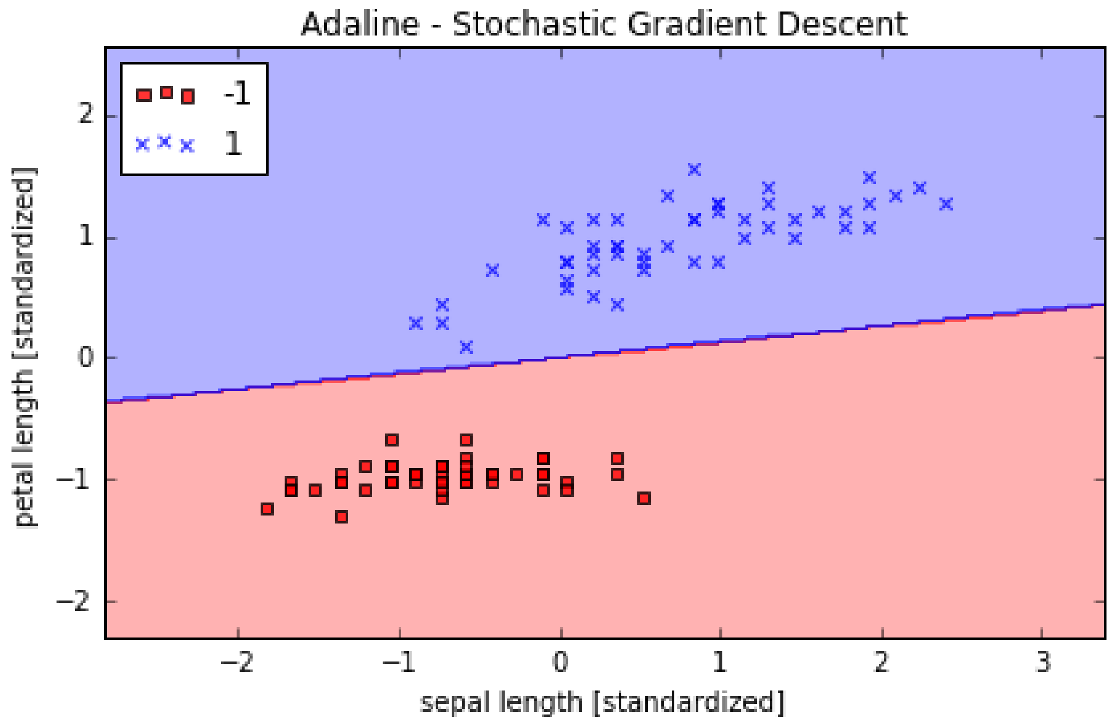
$$\Delta \mathbf{w}^{(i)} = \eta \left(y^{(i)} - \phi(z^{(i)}) \right) \mathbf{x}^{(i)}$$

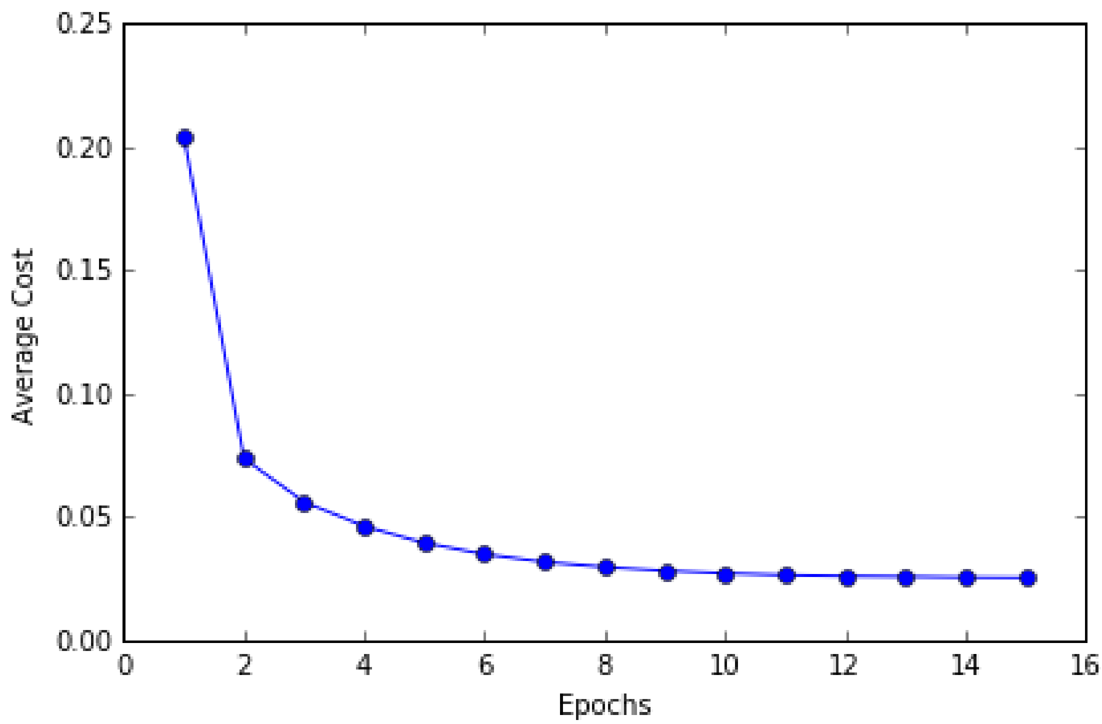
In [31]:
```python
class AdalineSGD(object):
    """ADAptive LInear NEuron classifier.

    Parameters
    ------------
    eta : float
      Learning rate (between 0.0 and 1.0)
    n_iter : int
      Passes over the training dataset.
    shuffle : bool (default: True)
      Shuffles training data every epoch if True to prevent cycles.
    random_state : int
      Random number generator seed for random weight
      initialization.


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


    """
    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):
        """ 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._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):
        """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):
        """Shuffle training data"""
        r = self.rgen.permutation(len(y))
        return X[r], y[r]

    def _initialize_weights(self, m):
        """Initialize weights to small random numbers"""
        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):
        """Apply Adaline learning rule to update the weights"""
```

```
        cost = 0
        ####################### BEGIN CODE ###############################
        # hint: obtain output of perceptron using the functions
        # self.net_input and self.activation
        # compute the errors using the output and obtain the Delta w
        # compute the overall cost J(w) to keep track of performance

        ####################### END CODE #################################

        return cost

    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 X

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

In [32]:
```
# apply stochastic gradient descent to
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()
```


Adaline - Stochastic Gradient Descent
-1
1
petal length [standardized]
sepal length [standardized]


Average Cost
Epochs