Implementation of Linear Classification Algorithms in Python: A Tutorial

Isa Ali
Department of Physics, Federal University Lokoja, P.M.B 1154, Lokoja, Nigeria,
Isa.ali@fulokoja.edu.ng

In this tutorial, we demonstrate the basic principle of the two machine learning algorithms for classifications: the *Perceptron* and *Adaline*. We achieved this by implementing these algorithms step by step in Python and provides the code snippets used to train them. The code snippets will boost our understanding of the basic linear classification algorithms, and their effective implementation in Python. This tutorial aims to lay the foundation for understanding advanced machine learning algorithms for classification such as support vector machine, logistic regression, and other regression models.

I. INTRODUCTION

We consider two simple Single-Layer Neural Networks: *Perceptron and Adaline*. The *Perceptron* algorithm is based on the widely known MCP neuron model [1]. It automatically learns the optimal weight coefficients that are needed to be multiplied with the input features in order to know whether or not a neuron will transmit signal (Figure. 1). It is used to predict the class a new data point belongs. The *Adaline* rule illustrates the basic concepts of defining and minimizing continuous function. The main difference between these algorithms is that, the weights are updated based on a linear activation function in *Adaline* rather than a unit step function like in the *Perceptron*.

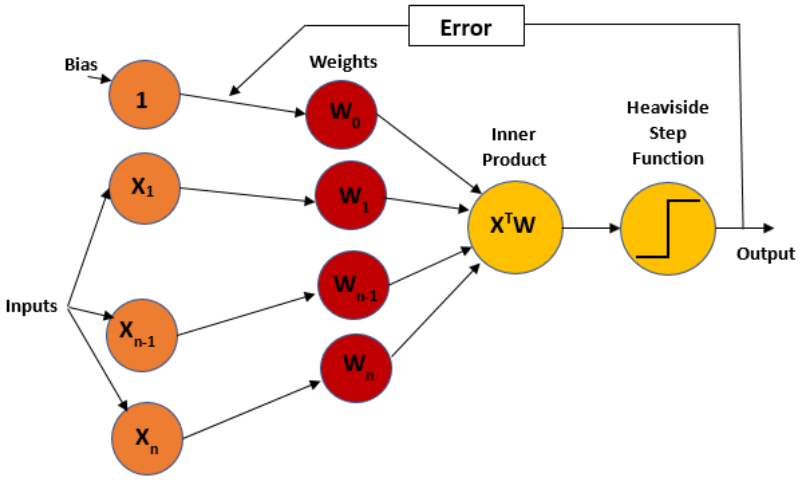

Figure 1: A schematic diagram illustrating the basic concept of the *Peceptron* algorithm

The *Perceptron* Model (Figure 1) receives a set of data inputs,$\boldsymbol{x}$, combines them with the weights, $\boldsymbol{w}$, to compute the inner product or net input. This inner product,$\boldsymbol{x^T w}$ is passed onto the **Heaviside step function** or **threshold function** which generates a binary output of -1 or +1 (i.e., the predicted class label of the input data). The T represent the transpose of the vector,$\boldsymbol{x}$.The output is used to calculate the error of the prediction during the learning rate, and thus the weights are updated.

Consider the implementation of this algorithm as we train it with **Iris dataset** using **Python**. We will also consider the object-oriented approach to define the interface of the perceptron as *Python Class*. This allows the initialization of new *Perceptron Objects* which can learn from the training data via a **fit** method and make some predictions via a separate **predict** method [2].

With this definition, we initialize Perceptron objects with predefined learning rate, η, the number of epochs, n_iter or the passes over the training dataset. We initialize the various weights to a vector,R^{n+1}, where n represents the number features or dimension in the dataset. We added 1 to the first element in this vector which represents the bias unit [2].

We will use the *Perceptron* algorithm for binary classification task and take two classes -1 (negative class) and +1 (positive class) into account. Let's also consider decision function $\phi(z)$ and takes into account a linear combination of input values, x, alongside weight vector, w, such that z is refers to as the net input or inner product.

$$z = w_1x_1 + w_2x_2 + \cdots + w_mx_m \tag{1}$$

$$w = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix}$$

$$x = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix}$$

When the net input for a certain example, $x^{(i)}$, becomes greater than a predefined threshold, θ, we can predict class -1 otherwise class 1. The decision function, $\phi(.)$, is taken as a unit step function.

such that:

$$\phi(z) = \begin{cases} 1 & if\ z \geq \theta, \\ -1 & otherwise. \end{cases} \tag{2}$$

Let's define weight-zero w_0 as $w_0 = -\theta$,

$x_0 = -1$ and z , the net input can be written as:

$$z = w_0 + w_1 + \cdots + w_m = \boldsymbol{w^T x} \tag{3}$$

and

$$\phi(z) = \begin{cases} 1 & if\ z \geq \theta, \\ -1 & otherwise. \end{cases} \tag{4}$$

The weight, $w_0 = -\theta$ is referred to as the **Bias unit** in **Machine Leaning** literature.

The **Perceptron Algorithm** can be summarized as follows [2]:

1. Initialize the weight to zero or some small random numbers.

2. For each training example, $x^{(i)}$:

(a) Compute the output value, $\hat{y}$.

(b) Update the weights.

Note: The value of the output, $\hat{y}$, is the class label predicted by the step function predefined above. The simultaneous update of each weight, w_j,in the weight vector, w, can be written as:

$$w_j := w_j + \Delta w_j \tag{5}$$

The updated value w_j is calculated as

$$\Delta w_j = \eta(y^{(i)} - \hat{y}^{(i)})x_j^{(i)} \tag{6}$$

The learning rate, η takes a constant value between 0.0 and 1.0, and $y^{(i)}$represents true class label of the ith training example, and $\hat{y}^{(i)}$ represent the **predicted class label**.

Note also that we update all the weights in the weight vector simultaneously and that the predicted label, $\hat{y}^{(i)}$ is not recomputed before all the weights are updated through the various update values, Δw_j.

The updates values for two-dimensional dataset can be written as follow:

$$\Delta w_0 = \eta(y^{(i)} - (output(i))x_j^{(i)} \tag{7}$$

$$\Delta w_1 = \eta(y^{(i)} - (output(i))x_1^{(i)} \tag{8}$$

$$\Delta w_2 = \eta(y^{(i)} - (output(i))x_2^{(i)} \tag{9}$$

The output is used to compute the error associated with the prediction during the learning phase and the weights are updated.

The **learning rate**, η , as well as the number of **epochs**, (n_iter), are the so-called hyperparameters (or tuning parameters) of the *Perceptron* and *Adaline* learning algorithms. Using these parameters, we can now initialize new *Perceptron* objects with a given learning rate, **eta**, and the number of **epochs**, n_iter (passes over the training dataset). Via the **fit** method, we initialize the weights in $\boldsymbol{self}.\boldsymbol{w}$_self to a vector, $\boldsymbol{R^{n+1}}$**,**where n stands for the number of dimensions (features) in the dataset, and we add 1 for the first element in this vector that represents the **bias unit**. Remember that the first element in this vector, $self.w$_[0], represents the so-called bias unit (for more information check out the code snippets in the Appendix).

After the weights have been initialized, the fit method loops over all individual examples in the training dataset, and updates the weights according to the perceptron learning rule (for more information check out the code snippets in the Appendix).

The predict method predicts the class labels, which is called in the fit method during the training of the datasets in order to get the class label for the weight updates. It can also be used to predict the class label of new dataset after fitting our model. Furthermore, the misclassifications of the class labels can be collected during each epoch in the $self.erros\_list$ in order to analyze the performance of our *Perceptron* model during the training process. The **np.dot** function that is used in the net input method simply calculates the vector dot product, $w^T\boldsymbol{x}$.

We limit the analysis of our *Peceptron* model to four feature variables; (I) sepal length and petal length and (II) sepal width and petal width to visualize the decision regions of the trained models in scatter plots. We also consider two classes for flowers, Setosa and Versicolor in the **Iris dataset.** The Iris Dataset can be accessed via this is web link:
UCI server at https:// archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data.

II. RESULTS AND DISCUSSIONS

For the Sepal length and petal length, we run the Perceptron codes given in **Appendix A**. We assigned them to a feature matrix, X *and* X_1 , which can be visualize through the 2D-scatterplots in Figures 2(a, b) below. The sepal length and petal length are assigned to X, while sepal width and petal width are assigned to X_1.

The distribution of flower examples in the **Iris Dataset** can observed in the two-dimensional scatter plots (Figures 2(a, b)). The scatter plot in Figure 2(a) shows the two features' axes: Petal length and sepal length, while Figure 2(b) shows the two feature axes: petal width and sepal width. From these plots, we can see that linear decision boundary can be used to differentiate Setosa flowers from Versicolor flowers. The *Perceptron*, been a linear classifier, can be used to classify the flowers in the Iris dataset perfectly.

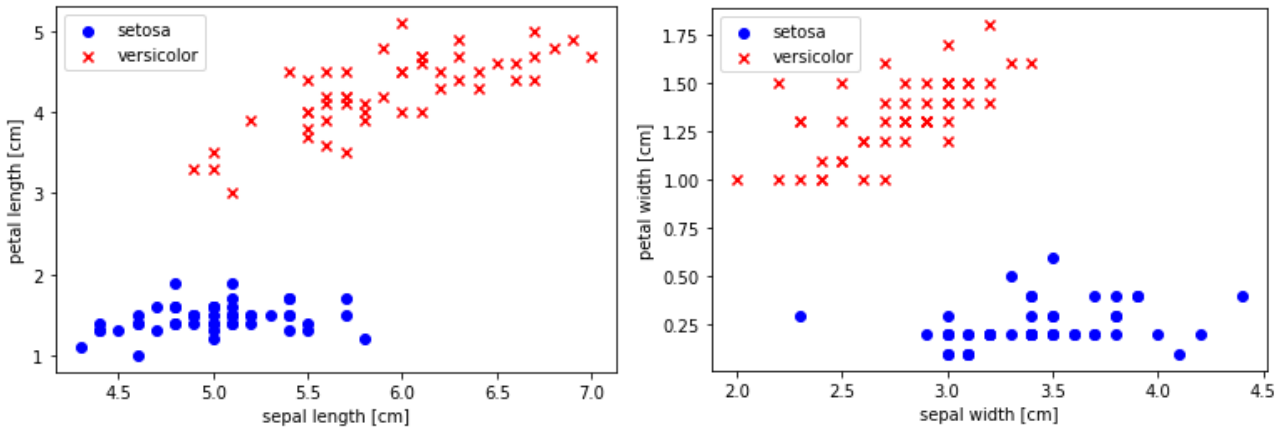

Figure 2(a): Scatterplot showing the distribution of flower examples in the Iris dataset along the two feature axes: petal length and sepal length. Figure 2(b): Scatterplot showing the distribution of flower examples in the Iris dataset along the two feature axes: petal width and sepal width.

To check whether our algorithm converges or not, we plot the misclassification error for each epoch and locate a decision boundary that will separate the two Iris flower classes.

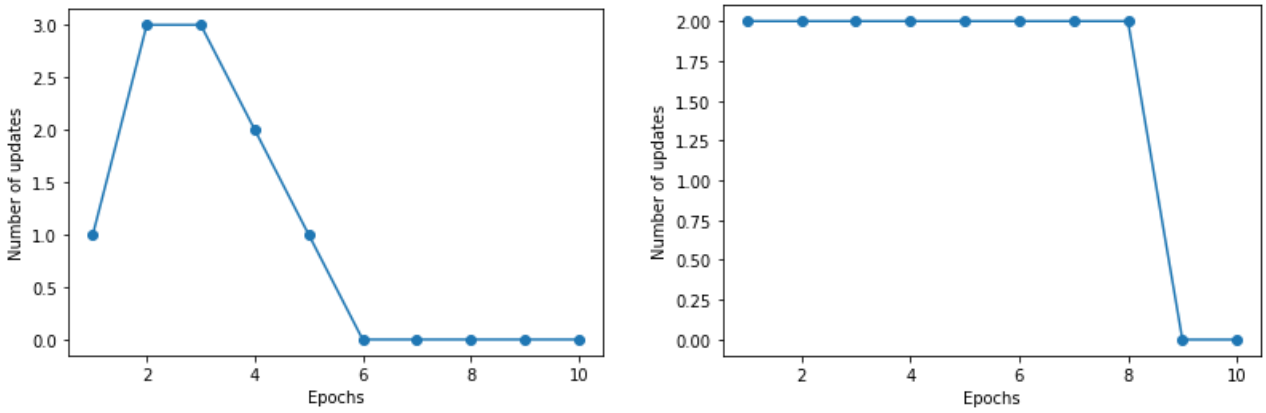

Figure 2(c): Plot of the misclassification errors versus the number of epochs for Iris dataset along the two feature axes: petal length and sepal length. Figure 2(d): Plot of the misclassification errors versus the number of epochs for Iris dataset along the two feature axes: petal width and sepal width**.**

After executing the python code (**Appendix B & C**), we can see the plots of miscalculation errors versus the number of epochs Figures 2(c, d). The **Perceptron** algorithm converge after the sixth and nineth epochs in both graphs, and thus classified our training examples perfectly.

To visualize the decision boundaries for the two-dimensional datasets, we execute a small convenience function (code snippets in the **Appendix C**. We draw contour plot via Matplotlib's contour function. This maps various decision regions to different colors for each predicted class in the grid array. The results are given in Figures 2(d, e) which show the plots of the decision regions. We can also observe that the resulting contour plots show that our *Peceptron* Algorithm learned a decision boundary that it uses to classify all of the flower examples in the **Iris data** subset perfectly.

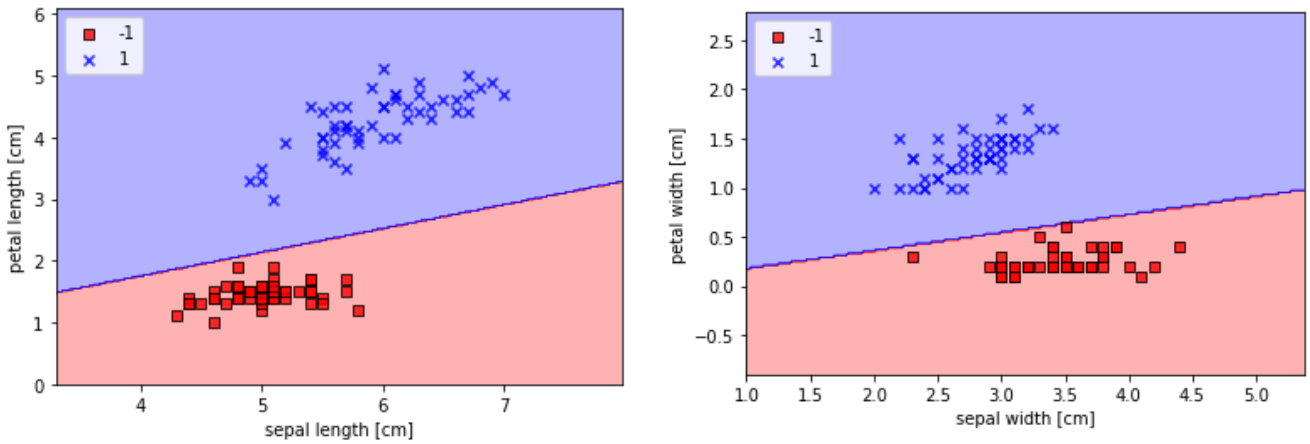

Figure: 2(e) A contour plot mapping the different decision regions to different colors for each predicted class in the two-dimensional grid array, Figure: 2(f) A contour plot mapping the different decision regions to different colors for each predicted class in the two-dimensional grid array.

III. ADAPTIVE LINEAR NEURON (ADALINE)

Let's consider another type of single-layer neural network (NN): Adaline [3]. It illustrates the key concepts of defining and minimizing continuous cost functions. The main difference between this and the *Perceptron* algorithm is that the weights are updated based on a linear **activation function** instead of using a **unit step function** like in the *Perceptron*. In *Adaline* figure 3, this linear **activation function**, $\phi(z)$, which is simply the **identity function** of the net input, such that:

$$\phi(w^T \boldsymbol{x}) = w^T \boldsymbol{x} \tag{10}$$

While the linear **activation function** is used for learning the weights, the **threshold function** is used to make the final prediction, which is similar to the unit step function.

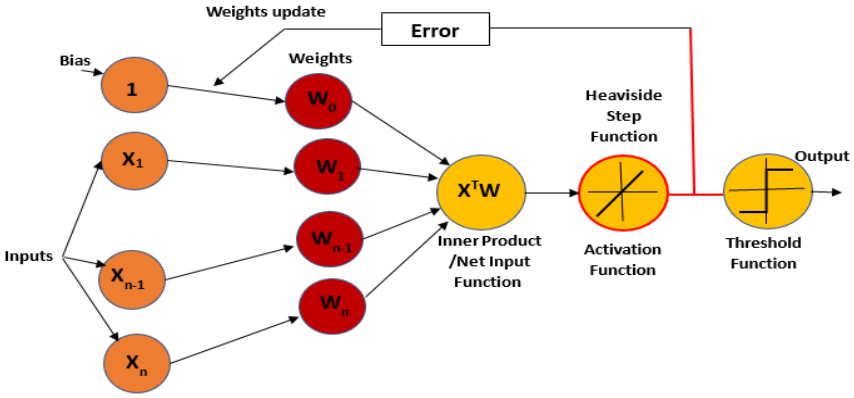

Figure 3: A schematic diagram illustrating the basic concept of the *Adaline* algorithm.

The **Adaline** model does compare the true class labels with the linear **activation function**'s continuous valued output to compute the **model error** and update the **weights** in Figure 3. Conversely, the **Perceptron** algorithm compares the true class labels to the predicted class labels.

IV. USING COST FUNCTION TO MINIMIZE COST FUNCTIONS

Optimizing the *objective function* is key to supervised machine learning algorithm, and is often taken as a cost function. In the case of *Adaline Algorithm*, we define a cost function, J, to learn the wights as the sum of the **squared errors** (SSE) between the true class label and the calculated outcome. The *activation function* is linear, continuous and has an edge over the unit step function, in that, it is differentiable. It is also convex in nature such that gradient descent (an optimization algorithm) can be used to locate the weights that minimize the cost function in order to classify the flowers in the **Iris Dataset**.

$$J(\boldsymbol{w}) = \frac{1}{2}\sum_i \left(y^{(i)} - \phi\ (z^{(i)})\right)^2 \tag{11}$$

$$\frac{\partial \mathrm{J}}{\partial w_j} = -\sum \left(y^{(i)} - \phi(z^{(i)})\right) x_j^{(i)} \tag{12}$$

The update of the weights,w_j, can be written as

$$\Delta w_j = -\eta \frac{\partial \mathrm{J}}{\partial w_j} = \eta \sum \left(y^{(i)} - \phi(z^{(i)})\right) x_j^{(i)} \tag{13}$$

The Adaline algorithm has updated weights as:

$$\mathrm{w} := \mathrm{w}\ + \Delta \mathrm{w} \tag{14}$$

And the Squared Error Derivative is given as**:**

$$\frac{\partial \mathrm{J}}{\partial w_j} = -\sum_{\boldsymbol{i}} \left(y^{(i)} - \phi(z^{(i)})\right) x_j^{(i)} \tag{15}$$

Even though Adaline algorithm looks identical to perceptron's, it is noteworthy that the weight update is calculated based on all examples in the training dataset instead of updating the weights incrementally after each training example (i.e., in *Peceptron's algorithm*). The *Adaline* learning rule is also referred to as **batch gradient descent.**

V. IMPLEMENTATION OF ADALINE ALGORITHM IN PYTHON

To implement the *Adaline algorithm* in Python, we change the **fit** method in the *Perceptron* code provided in the **Appendix F**. The weights are updated by the minimizing the cost function using gradient descent discussed in the previous session. We compute the gradient based on the whole training dataset using self.**eta***errors. **Sum** () for the zero-weight (**bias unit**), and via selt.**eta***__**X**__.__**T**__.__**dot** (errors) for the weights 1 to m, where **X**.**T**.**dot** (errors) represents the matrix-vector multiplication between the error vector and the feature matrix.

As we can see in the resulting cost-function plots, we encountered two different of problem. The left chart in Figure 4(a) shows what could happen if we choose a learning rate, $\eta = 0.01$**,** that is too large. Instead of minimizing the cost function, the error becomes larger in every epoch, because we overshoot the global minimum. Conversely, in Figure 4(b), we can observe that the cost function decreases and the errors become smaller for every epoch with a learning rate of $\eta = 0.0001$**.** This learning rate will require us to have large number of epochs to converge to the global cost minimum.

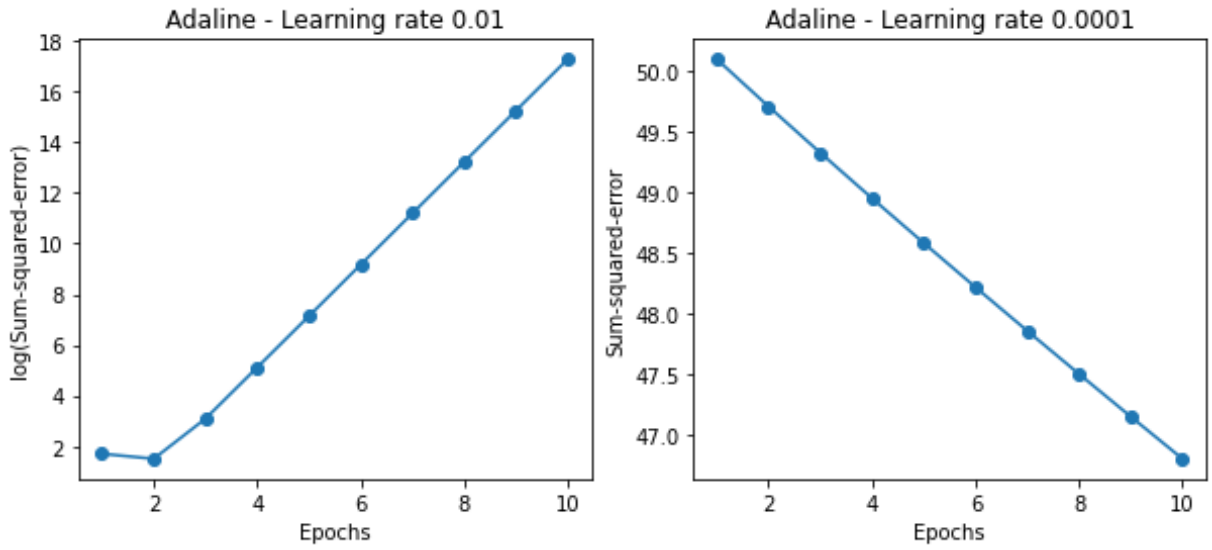

Figure: 4(a) illustrates the case of choosing a learning rate that is too large $\eta = 0.01$ and the errors become larger in every epoch. Figure: 4(b) illustrates the case of a well-chosen learning rate, $\eta = 0.0001$, where the cost decreases gradually, moving in the direction of the global minimum.

To visualize the decision regions using the Adaline codes provided in **Appendix A**, we execute the code snippets and obtained the results in Figures 4(c, d) given as follow.

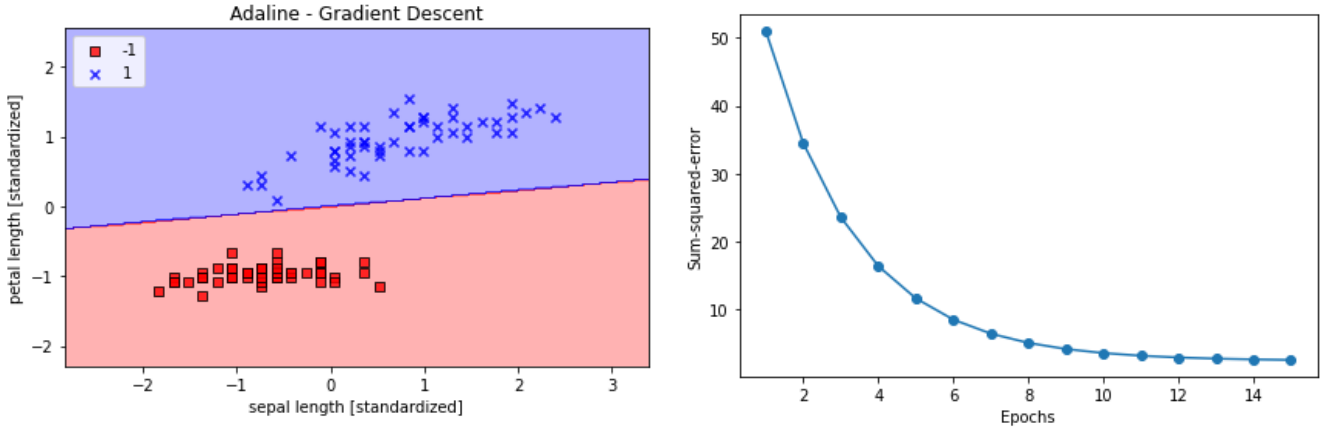

Figure 4(c) A contour plot mapping the different decision regions in the two-dimensional grid array using Adaline Algorithm. Figure 4(d): The Plot of a declining cost function.

We draw contour plot in Figure 4(a) via Matplotlib's contour function. This map shows various decision regions to different colors for each predicted class in the grid array. We can observe that, the resulting contour plots show that our **Adaline Algorithm** learned a decision boundary that was used to classify all the flower examples in the **Iris data** subset perfectly.

As we can see in the plot in Figure 4(d), **Adaline** has now converged after training on the standardized features using a learning rate of η = 0.01. However, note that the SSE remains non-zero even though all flower examples were classified correctly (Figure 4(d)).

VI. CONCLUSION

We demonstrated the basic principle of linear classification algorithms in Python with a special focus on the Perceptron and Adaline algorithms. The Iris dataset was used to train these algorithms, and Spyder Interactive Development Environment was used to implement the Python codes. We also explore how the tuning parameters, η and epochs, affect the cost function, and the convergence of both algorithms. In overall, the algorithms learned the decision boundary and classified all the flower examples in the **Iris dataset** perfectly. This brief tutorial aims to provide the foundation for understanding single-layer Neutral Network in preparation for more sophisticated Multi-layer Neural Networks.

---

[1] Rosenblatt, F ,1957, *The Perceptron: A Perceiving and Recognizing Automaton*, Cornell Aeronautical Laboratory.

[2] Sebastian, R., and Vahid, M., 2019, *Machine Learning and Deep Learning with Python*, *Scikit-Learn, and Tensor Flow*, (Packt Publishing, www.packt.com).

[3] Widdrow, B., 1960, *An Adaptive "Adaline" Neuron Using Chemical "Memistors"*, Technical Report Number 1553-2, Stanford Electron Labs, Stanford, CA.

Appendix A: Code snippets used to load the Iris Dataset and generate the scatter plot in Figure 2(a).

```
"""
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import csv
from matplotlib import pyplot
from pandas import read_csv
from sklearn.model_selection import train_test_split
from sklearn.linear_model import Perceptron

s='iris.data'
df = pd.read_csv(s,header=None, encoding='utf-8')

# filename = 'iris.csv'
# names = ['sepal-length', 'sepal-width', 'petal-length', 'petal-width', 'class']
# dataset = read_csv(filename, names=names)



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

Appendix B: Code snippets used to generate plot in Figure 2(c)

```
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):

        """Fit training data.
        Parameters
        ----------
        X : {array-like}, shape = [n_examples, n_features]
        Training vectors, where n_examples is the number of
        examples and n_features is the number of features.
        y : array-like, shape = [n_examples]
        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.errors_ = []

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

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

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

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()
```

Appendix C: Code Snippets used to generate the contour map in Figure 2(e).

```
from matplotlib.colors import ListedColormap
def plot_decision_regions(X, y, classifier, resolution=0.02):
    # setup marker generator and color map
    markers = ('s', 'x', 'o', '^', 'v')
    colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
    cmap = ListedColormap(colors[:len(np.unique(y))])

    # plot the decision surface
    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution),
    np.arange(x2_min, x2_max, resolution))
    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 examples
    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='b

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

Appendix D: Code Snippets used to generate the contour map in Figure 2(b) and 2(d).

```
######### for the Flowers Widths#######

#select setosa and versicolor
y1 = df.iloc[0:100, 4].values
y1 = np.where(y1 == 'Iris-setosa', -1, 1)
# extract sepal width and petal width
X1 = df.iloc[0:100, [1, 3]].values
# plot data
plt.scatter(X1[:50, 0], X1[:50, 1], color='brown', marker='o', label='setosa')
plt.scatter(X1[50:100, 0], X1[50:100, 1],color='purple', marker='x', label='versicolor')
plt.xlabel('sepal width [cm]')
plt.ylabel('petal width [cm]')
plt.legend(loc='upper left')
plt.show()



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):
            self.eta = eta
            self.n_iter = n_iter
            self.random_state = random_state
```

```
    def fit(self, X1, y1):
        """Fit training data.
        Parameters
        ----------
        X1 : {array-like}, shape = [n_examples, n_features]
        Training vectors, where n_examples is the number of
        examples and n_features is the number of features.
        y1 : array-like, shape = [n_examples]
        Target values.
        Returns
        -------
        self : object
        """
        rgen = np.random.RandomState(self.random_state)
        self.w_ = rgen.normal(loc=0.0, scale=0.01,size=1 + X1.shape[1])
        self.errors_ = []

        for _ in range(self.n_iter):
            errors = 0
            for xi, target in zip(X1, y1):
                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, X1):
        """Calculate net input"""
        return np.dot(X1, self.w_[1:]) + self.w_[0]

    def predict(self, X1):
        """Return class label after unit step"""
        return np.where(self.net_input(X1) >= 0.0, 1, -1)

ppn = Perceptron(eta=0.1, n_iter=14)
ppn.fit(X1, y1)
plt.plot(range(1, len(ppn.errors_) + 1),ppn.errors_, marker='o')
plt.xlabel('Epochs')
plt.ylabel('Number of updates')
plt.show()
```

Appendix E: Code Snippets used to generate the contour map in Figure 2(f).

```
from matplotlib.colors import ListedColormap
def plot_decision_regions(X1, y1, classifier, resolution=0.02):
    # setup marker generator and color map
    markers = ('s', 'x', 'o', '^', 'v')
    colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
    cmap = ListedColormap(colors[:len(np.unique(y1))])

    # plot the decision surface
    x1_min, x1_max = X1[:, 0].min() - 1, X1[:, 0].max() + 1
    x2_min, x2_max = X1[:, 1].min() - 1, X1[:, 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 examples
    for idx, cl in enumerate(np.unique(y1)):
        plt.scatter(x=X1[y == cl, 0],y=X1[y == cl, 1],alpha=0.8,c=colors[idx],marker=markers[idx],label=cl,edgecolor='black')

plot_decision_regions(X1, y1, classifier=ppn)
plt.xlabel('sepal width [cm]')
plt.ylabel('petal width [cm]')
plt.legend(loc='upper left')
plt.show()
```

Appendix F: Code Snippets used to generate the contour map in Figures 4(a - d).

```
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):
        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_examples, n_features]
        Training vectors, where n_examples
        is the number of examples and
        n_features is the number of features.
        y : array-like, shape = [n_examples]
        Target values.
        Returns

        -------
        self : object

        """
        import numpy as np
        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):
        """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)


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()


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, classifier=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()
```