**Soft Computing**

**Lab – 2**

1. **Adaline Master**

**Source Code :**

**Main.py**

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from adalinegd import AdalineGD

from adalinesgd import AdalineSGD

import pdr

# get the iris data

df = pd.read\_csv('https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data',

    header = None)

# Plot 100 samples of the data

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('petal length')

plt.ylabel('sepal length')

plt.legend(loc = 'upper left')

plt.show()

# Standardize the data

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()

# Create the AdalineGD model

model1 = AdalineGD(n\_iter = 15, eta = 0.01)

# Train the model

model1.fit(X\_std, y)

# Plot the training error

plt.plot(range(1, len(model1.cost\_) + 1), model1.cost\_, marker = 'o', color = 'red')

plt.xlabel('Epochs')

plt.ylabel('Sum-squared-error')

plt.show()

# Plot the decision boundary

pdr.plot\_decision\_regions(X\_std, y, classifier = model1)

plt.title('Adaline - Gradient Descent')

plt.xlabel('sepal length [standardized')

plt.ylabel('petal length [standardized]')

plt.legend(loc = 'upper left')

plt.show()

# Create the AdalineSGD model

model2 = AdalineSGD(n\_iter = 15, eta = 0.01, random\_state = 1)

# Train the model

model2.fit(X\_std, y)

# Plot the training errors of both of the models

plt.plot(range(1, len(model2.cost\_) + 1), model2.cost\_, marker = 'x', color = 'blue')

plt.xlabel('Epochs')

plt.ylabel('Sum-squared-error')

plt.show()

# Plot the decision boundary

pdr.plot\_decision\_regions(X\_std, y, classifier = model2)

plt.title('Adaline - Stochastic Gradient Descent')

plt.xlabel('sepal length [standardized')

plt.ylabel('petal length [standardized]')

plt.legend(loc = 'upper left')

plt.show()

**adalinesgd.py**

import numpy as np

from numpy.random import seed

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.

    Attributes

    -----------

    w\_ : 1d-array

        Weights after fitting.

    errors\_ : list

        Number of misclassifications in every epoch.

    shuffle : bool (deafult: True)

        Shuffles training data every epoch if True

        to prevent cycles.

    random\_state : int (default: None)

        Set random state for shuffling and

        initializing the weights.

    """

    def \_\_init\_\_(self, eta = 0.01, n\_iter = 10, shuffle= True,

                random\_state = None):

        self.eta = eta

        self.n\_iter = n\_iter

        self.w\_initialization = False

        self.shuffle = shuffle

        if random\_state:

            seed(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.

        Return

        -------

        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 = np.random.permutation(len(y))

        return X[r], y[r]

    def \_initialize\_weights(self, m):

        """ Initialize weights to zeros """

        self.w\_ = np.zeros(1 + m)

        self.w\_initialized = True

    def \_update\_weights(self, xi, target):

        """ Apply Adaline learning rule to update the weights """

        output = self.net\_input(xi)

        error = (target - output)

        self.w\_[1:] += self.eta \* xi.dot(error)

        self.w\_[0] += self.eta \* error

        return 0.5 \* (error \*\* 2)

    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 self.net\_input(X)

    def predict(self, X):

        """ Return class label after the unit step """

        return np.where(self.activation(X) >= 0.0, 1, -1)

**adalinegd.py**

import numpy as np

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.

    Attributes

    -----------

    w\_ : 1d-array

        Weights after fitting.

    errors\_ : list

        Number of misclassifications in every epoch.

    """

    def \_\_init\_\_(self, eta = 0.01, n\_iter = 50):

        self.eta = eta

        self.n\_iter = n\_iter

    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.

        Return

        -------

        self : object

        """

        self.w\_ = np.zeros(1 + X.shape[1])

        self.cost\_ = []

        for i in range(self.n\_iter):

            output = self.net\_input(X)

            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 self.net\_input(X)

    def predict(self, X):

        """ Return class label after unit step """

        return np.where(self.activation(X) >= 0.0, 1, -1)

**pdr.py**

import numpy as np

import matplotlib.pyplot as plt

from matplotlib.colors import ListedColormap

""" Function to color the deciosion regions """

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 = 0.4, cmap = cmap)

    plt.xlim(xx1.min(), xx1.max())

    plt.ylim(xx2.min(), xx2.max())

    # plot class samples

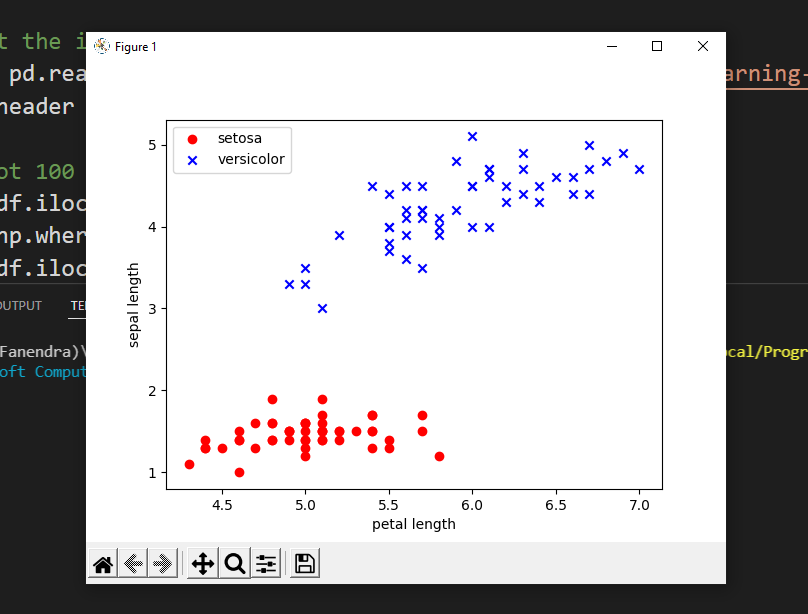
    for idx, cl in enumerate(np.unique(y)):

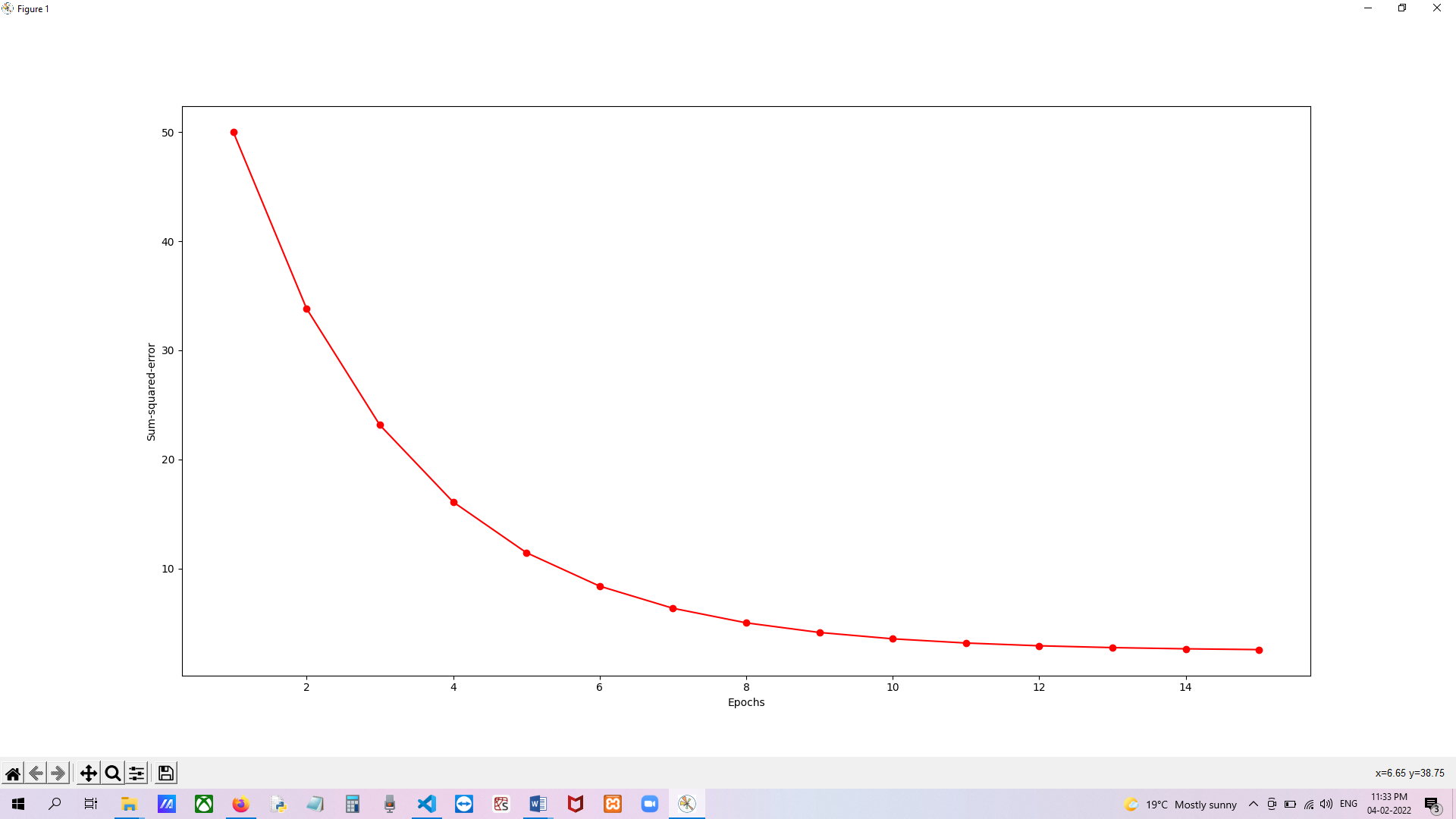
        plt.scatter(x = X[y == cl, 0], y = X[y == cl, 1],

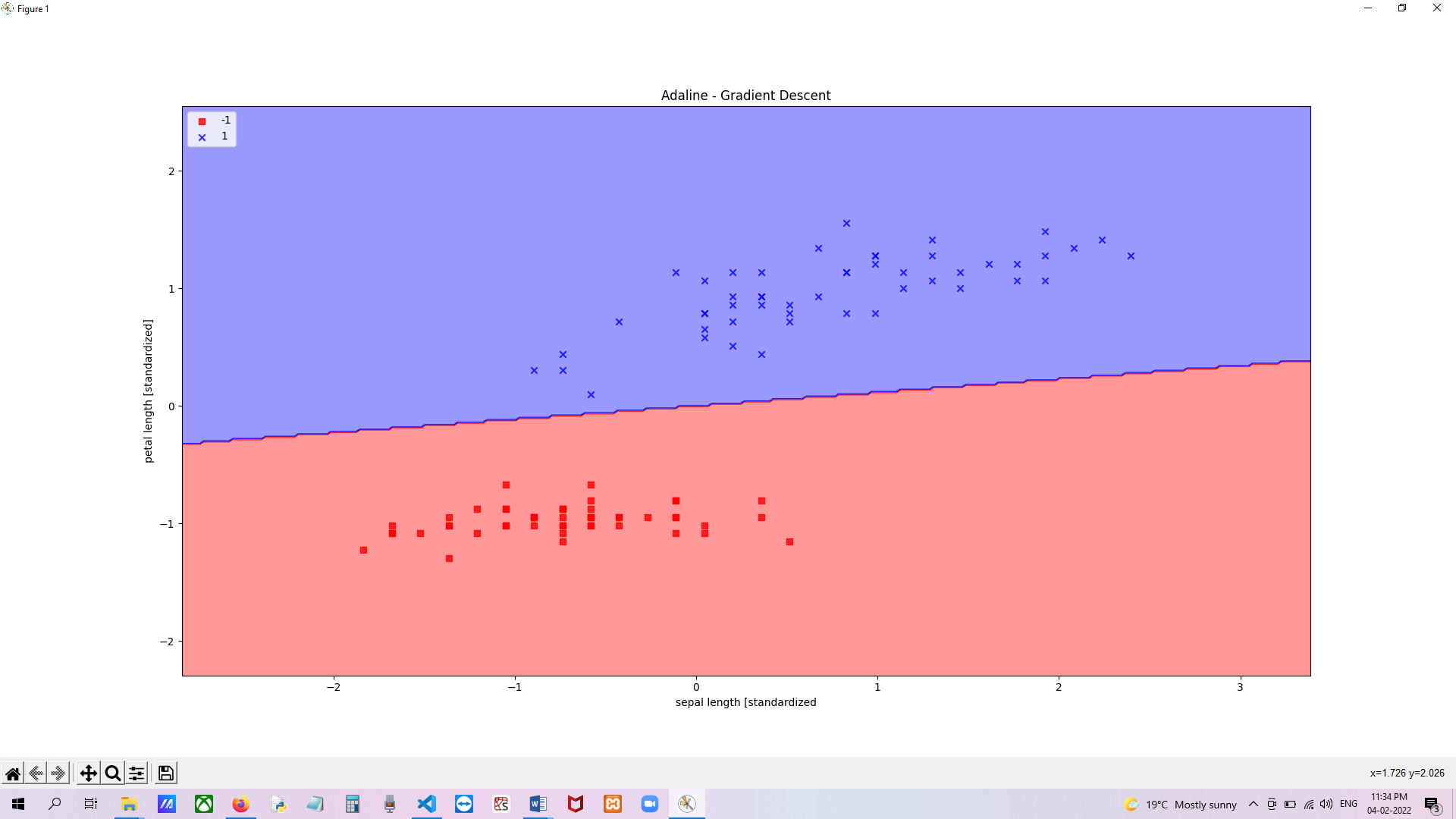
            alpha = 0.8, c = cmap(idx), marker = markers[idx],

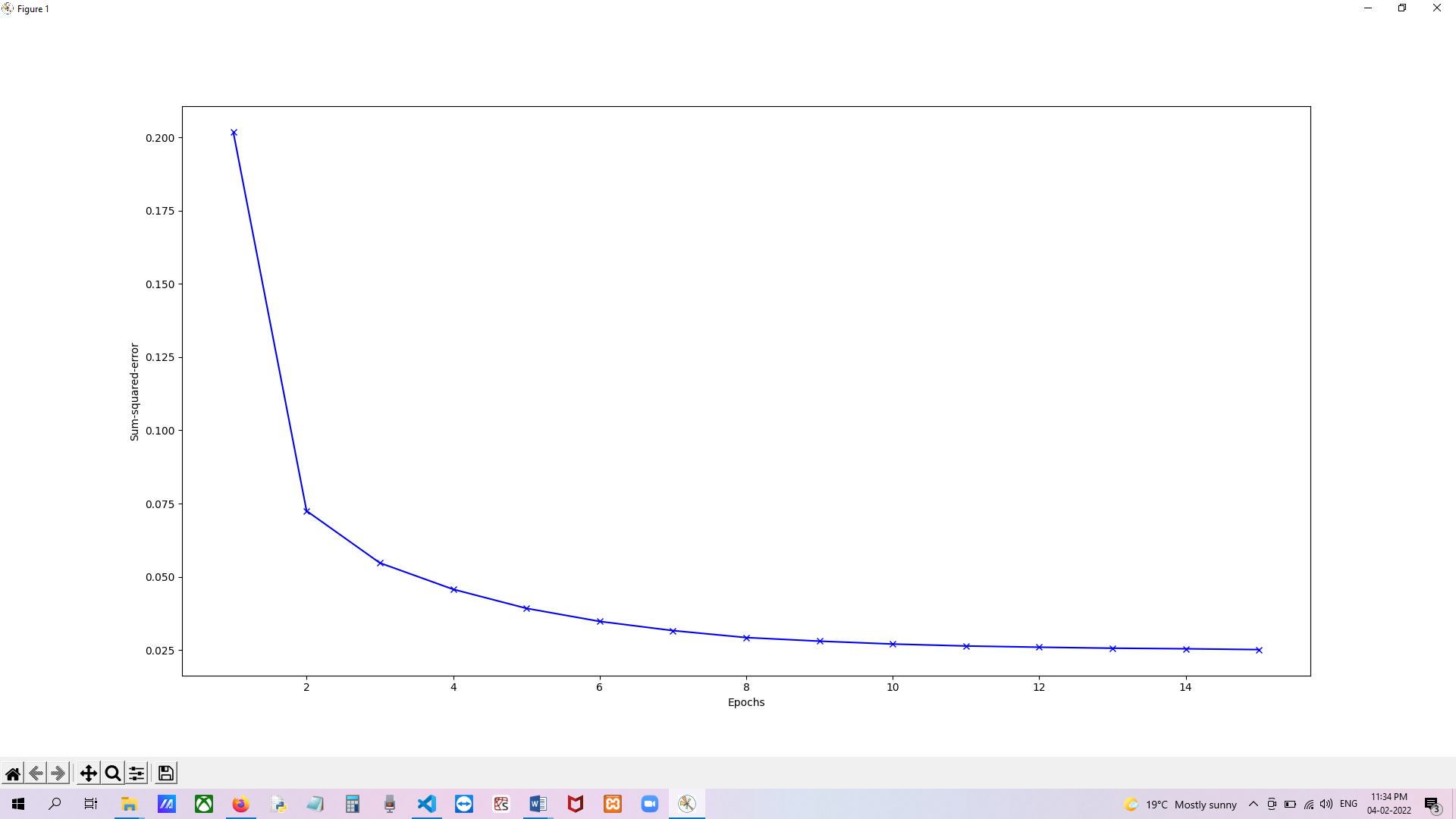
            label = cl)

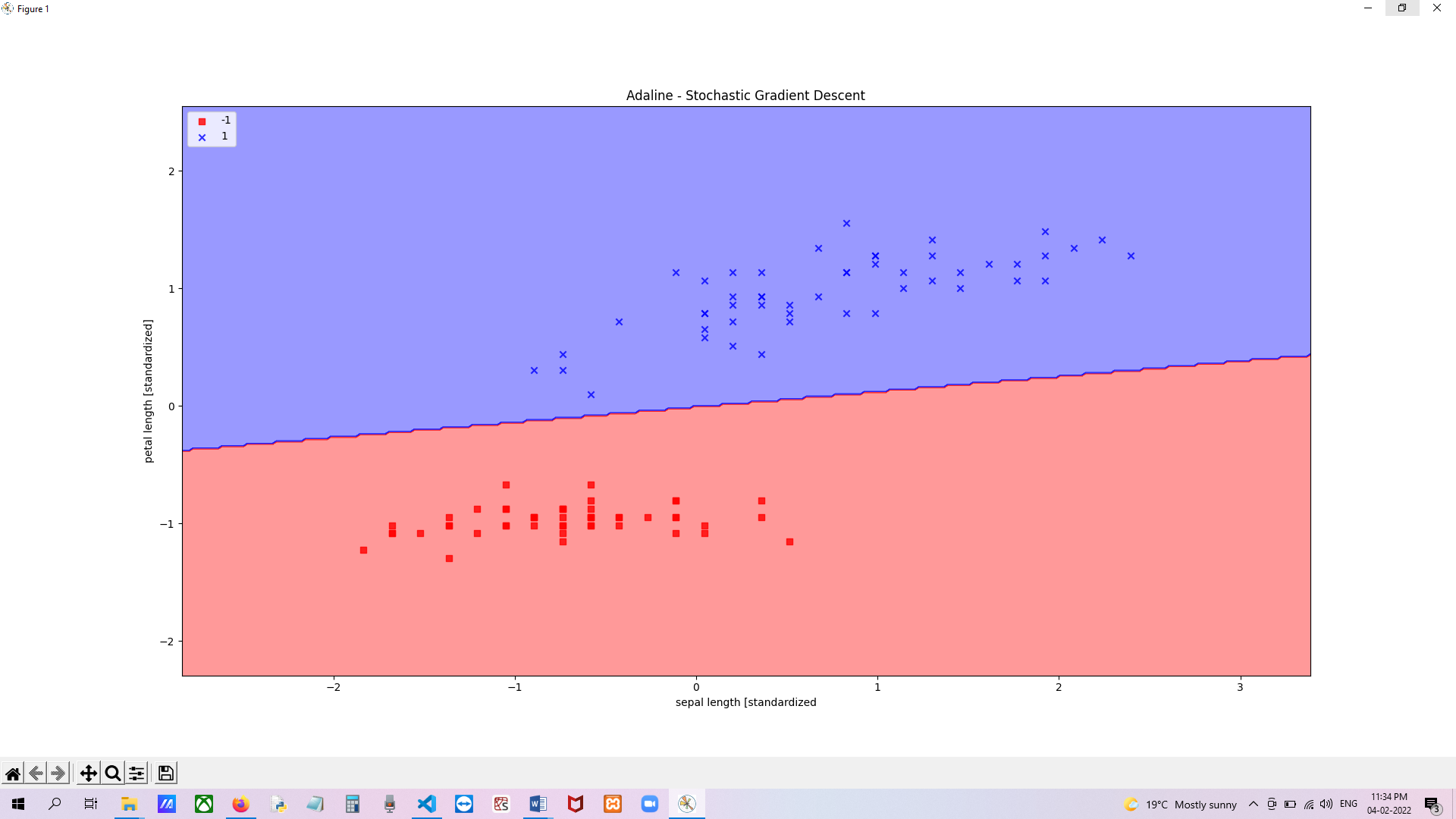
**Output :**

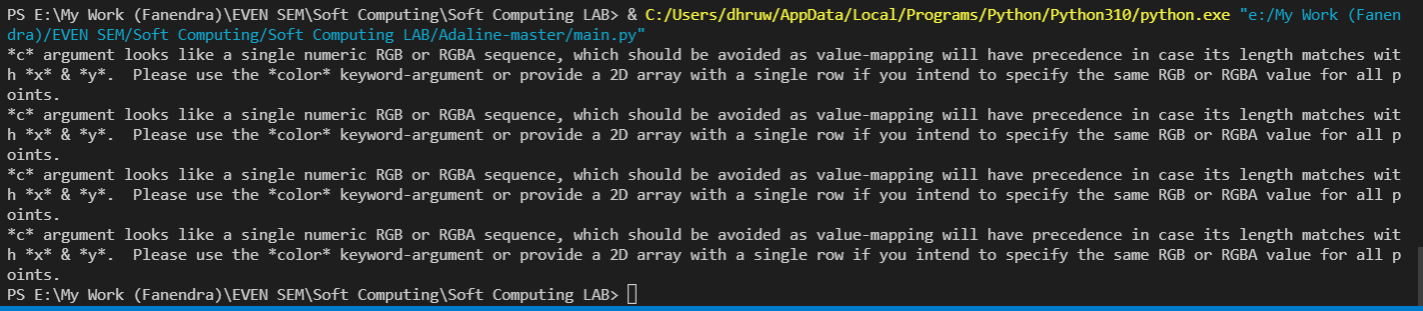
****









****