```python
import numpy as np
import os
import pandas as pd
import matplotlib.pyplot as plt
class Reservoir:

   def __init__(self, input_neurons, reservoir_neurons):
       self.inputs = input_neurons
       self.outputs = input_neurons
       self.reservoir_size = reservoir_neurons
       self.calc_weights(reservoir_neurons, input_neurons,
input_neurons)

   def calc_weights(self,reservoir_size,outputs,inputs):
       self.weights_in = np.random.normal(0, np.sqrt(0.002),
size=(reservoir_size, inputs))
       self.weights_r = np.random.normal(0,
np.sqrt(2/reservoir_size), size=(reservoir_size, reservoir_size))
       self.weights_out = np.empty((outputs, reservoir_size))

   def ridge_regression(self, x, y, k):
       T = x.shape[1]
       r = np.zeros((self.reservoir_size, T + 1))
       for t in range(0, T):
           r[:, t + 1] = np.tanh(self.weights_r @ r[:, t] +
self.weights_in @ x[:, t])
       # Remove initial to avoid fluctuations
       r = r[:, 26:]
       y = y[:, 25:]
       # Using rigid regression to get the output weights
       self.weights_out = y @ r.T @ np.linalg.inv(r @ r.T + k *
np.identity(self.reservoir_size))

   def predict(self, x, steps):
       r = np.zeros((self.reservoir_size,))
       for i in range(x.shape[1]):
           r = np.tanh(self.weights_r @ r + self.weights_in @ x[:,
i])
       reservoir_output = np.empty((self.outputs, steps))
       for j in range(steps):
           reservoir_output[:,j] = self.weights_out @ r
           r = np.tanh(self.weights_r @ r + self.weights_in @
reservoir_output[:, j])
       return reservoir_output

def read_data(file_name: str) -> np.ndarray:
   return pd.read_csv(file_name, delimiter=',', header=None).values
```

```python
def main():
    reservoir_computer = Reservoir(3, 500)
    # training the data with ridge regression
    training_set = read_data('training-set.csv')
    x = training_set[:, :-1]
    y = training_set[:, 1:]
    k = 0.01
    reservoir_computer.ridge_regression(x, y, k)
    #  feeding the test data through the network
    test_set = read_data('test-set-2.csv')
    steps = 500
    test_pred = reservoir_computer.predict(test_set, steps)
    # plot training set and predicted test set
    fig = plt.figure()
    plot_data = plt.axes(projection='3d')
    x, y, z = test_set
    plot_data.plot3D(x, y, z, c='b',label='Actual test-set')
    x2, y2, z2 = test_pred
    plot_data.plot3D(x2, y2, z2, c='g', label='Predicted test-set')
    plt.legend()
    plt.show()
    np.savetxt('prediction.csv', test_pred[1, :], delimiter=',')

if __name__ == '__main__':
    os.chdir(os.path.dirname(__file__))
    main()
```

```
import matplotlib.pyplot as plt
import matplotlib.patches as mp
import os
import numpy as np
import pandas as pd
from tqdm import tqdm

class SelfOrganisingMap:
   def __init__(self, input_dimensions, output_shape):
       self.w = np.random.random((*output_shape,
input_dimensions))
       self.output_array = output_shape

   def initialize(self, x):
       dist = np.sum((self.w - x)**2, axis=-1)
       return np.unravel_index(np.argmin(dist), self.output_array)

   def neighbourhood_function(self, i, i0, n):
       return np.exp(-np.sum( (np.array(i) - np.array(i0))**2 ) /
(2 * n ** 2))

   def train(self, x, learnrate, n):
       i0 = self.initialize(x)
       deltaw = np.empty_like(self.w)
       for i in range(self.output_array[0]):
           for j in range(self.output_array[1]):
               deltaw[i,j,:] = learnrate *
self.neighbourhood_function((i, j), i0, n) * (x - self.w[i, j, :])
       self.w += deltaw

def read_data(file_name):
   return pd.read_csv(file_name, delimiter=',', header=None).values

def standardize(data):
   return data / np.max(data)

def train(som, data, epochs, initial_learnrate, learnrate_decay,
n, n_decay):
   b_size, _ = data.shape
   for epoch in tqdm(range(epochs)):
       learnrate = initial_learnrate * np.exp(-learnrate_decay *
epoch)
       n1 = n * np.exp(-n_decay * epoch)
       for _ in range(b_size):
           x = data[np.random.randint(b_size), :]
           som.train(x, learnrate, n1)
   return som
```

```python
def plot(neurons, targets, shape, a_plot,iris_flowers, colors):
   img = np.zeros((*shape, 4))
   for (label,), (x, y) in zip(targets, neurons):
       iris = iris_flowers[label]
       img[x, y] = colors[iris]
   a_plot.imshow(img, origin='lower')

def main():
   iris_data = read_data('iris-data.csv')
   iris_labels = read_data('iris-labels.csv')
   iris_data = standardise(iris_data)
   output_array = (40, 40)
   epochs = 10
   input_dimensions = 4
   som = SelfOrganisingMap(input_dimensions, output_array)
   # Using the initial weights
   map1 = np.array([som.initialize(x) for x in iris_data])
   # Train to get weights to make a new map
   initial_learnrate = 0.1
   learnrate_decay = 0.01
   n = 10
   n_decay = 0.05
   train(som, iris_data, epochs,
initial_learnrate,learnrate_decay, n, n_decay)
   map2 = np.array([som.initialize(x) for x in iris_data])
   # Making the plots
   iris_flowers = {
       0.0: 'Iris Setosa',
       1.0: 'Iris Versicolour',
       2.0: 'Iris Virginica',}
   colors = {
       'Iris Setosa': (1.0, 0.0, 0.0, 1.0),  # red
       'Iris Versicolour': (0.0, 1.0, 0.0, 1.0),  # green
       'Iris Virginica': (0.0, 0.0, 1.0, 1.0),}  # blue
   fig, (plot1, plot2) = plt.subplots(1, 2)
   plot(map1, iris_labels, output_array, plot1, iris_flowers,
colors)
   plot(map2, iris_labels, output_array, plot2, iris_flowers,
colors)
   legends = [mp.Patch(color=colors[iris], label=iris) for iris in
iris_flowers.values()]
   fig.legend(handles=legends, loc='lower center',
bbox_to_anchor=(0.6, 0.9), ncol=len(iris_flowers))
   plt.title('After learning ')
   plt.show()

if __name__ == '__main__':
   os.chdir(os.path.dirname(__file__))
```

```
main()
```

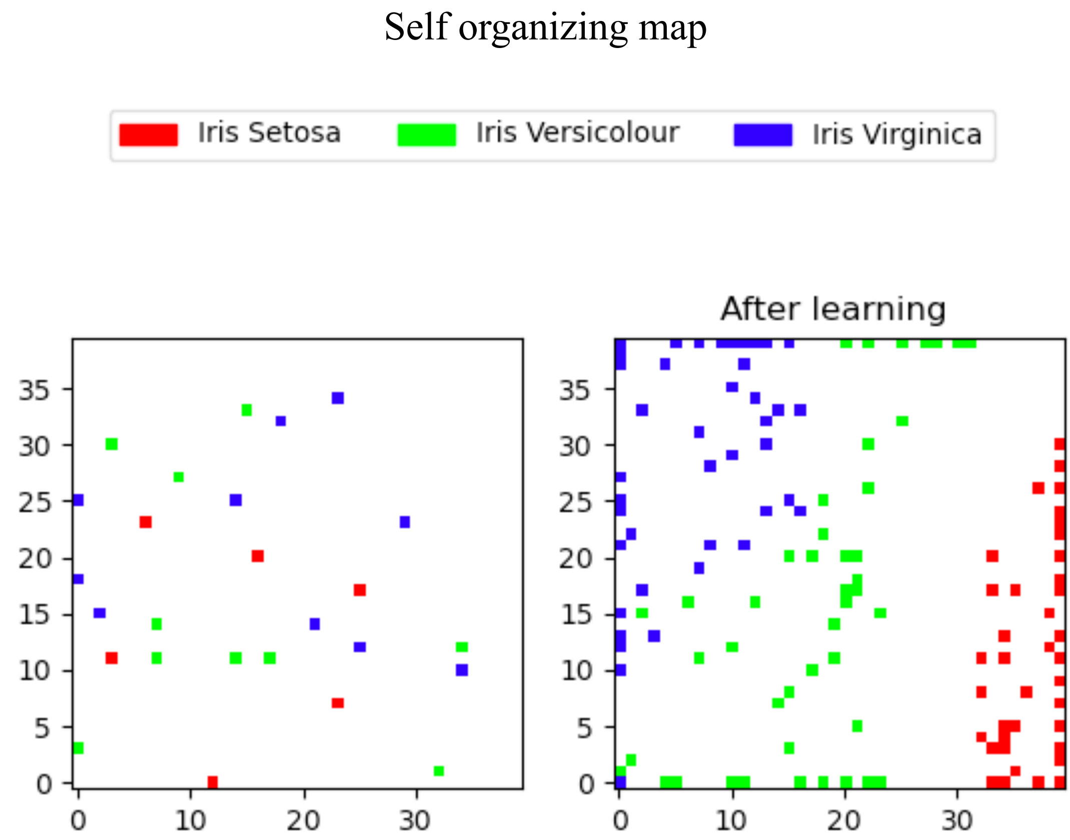

Figure 1 Plots of the location of the winning neuron in the output array.

As one can see in figure 1 the final plot consists of two panels, the left one shows the location of the winning neuron color-coded according to one of the three classes with randomly chosen initial weights. The right panel shows the same but the only difference is that the weights that have been used are the ones obtained after iterating the learning rule.

By analyzing the plots one can conclude that there is a significant difference. The left plot shows the winning neurons locating randomly which is understandable because the weights are initialized randomly and there is no training beforehand The left plot however shows the winning neurons located as three clusters corresponding to the three different types of Iris flowers, so after training the network by iterating the learning rule, the weight with the shortest distance gets rewarded which then is used to map the neurons which explains the clusters in the plot. The plot also shows that the patterns that are close to each other have corresponding neurons also close to each other. The class Iris Setosa with the red neurons is most separated from the other classes, which shows that that class is most different from the other classes. The fact that the other classes have some overlapped neurons may partly be due to the fact that they are similar but also errors in classifications.

One thing that initially concerned me was that my self organizing map was clustering the data points differently each time when running the program but I later realized that the reason behind was the random initialization of the weight matrix.