

Project5

1. Team member

- m5268101 Liu Jiahe
- m5251140 Ken Sato
- m5271051 Keisuke Utsumi

2. Team Project V

- Using SOFM(Self-Organization feature map) algorithm to cluster the Iris dataset (<http://www.ics.uci.edu/~mlearn/MLRepository.html>).

3. Algorithm

SOFM算法可以降低数据的维度同时保存数据之间空间上的拓扑结构。该算法中神经元二维网络分布，每个神经元有六个邻域，成六边形网格。每个神经元的权重数量和输入数据的向量维度相同。该算法步骤如下：

1. 初始化：
 - 初始化连接权值矩阵 w , 其中 $w[n][i]$ 表示第 n 个神经元与输入向量的第 i 个分量之间的连接权值。
2. 自组织映射循环：
 - a. 选择一个训练样本向量：
 - 从训练集中随机选择一个样本向量 $x[p]$ 。
 - b. 寻找最相似的神经元：
 - 计算样本向量 $x[p]$ 与每个神经元的距离。
 - 找到与样本向量距离最近的神经元, 即计算最小欧氏距离：
$$d(m_1, m_2) = \sum_{i=1}^I (w[m_1, m_2][i] - x[p][i])^2$$
, 其中 m_1 和 m_2 表示神经元在二维网格上的位置。
 - c. 更新邻域内的神经元权值：
 - 定义学习率 r 和邻域半径 nc , 它们随着迭代次数的增加而逐渐减小。
 - 对于每个神经元 $w[m_1, m_2]$, 计算其在二维空间上的位置 (x_1, x_2) , 偶数行横坐标加0.5是为了视觉上形成六边形网络, 但是不会影响神经元的权重
 - 如果神经元 (m_1, m_2) 位于最近神经元 (m_{10}, m_{20}) 的邻域内 (距离小于等于邻域半径 nc)：
 - 更新连接权值:

$w[m_1, m_2][i] = w[m_1, m_2][i] + r \cdot (x[p][i] - w[m_1, m_2][i])$, 其中 i 表示输入向量的分量索引。

3. 标定阶段:

- 将每个输入模式的的标签分配给最相似(距离最近)的神经元，并更新神经元的标签。

4. 打印结果:

- 打印最终神经元的标签矩阵，展示每个神经元的分类结果。

Iris数据集结构

```
1 5.1,3.5,1.4,0.2,Iris-setosa
2 4.9,3.0,1.4,0.2,Iris-setosa
3 4.7,3.2,1.3,0.2,Iris-setosa
4 4.6,3.1,1.5,0.2,Iris-setosa
5 5.0,3.6,1.4,0.2,Iris-setosa
6 5.4,3.9,1.7,0.4,Iris-setosa
7 4.6,3.4,1.4,0.3,Iris-setosa
8 5.0,3.4,1.5,0.2,Iris-setosa
9 4.4,2.9,1.4,0.2,Iris-setosa
10 4.9,3.1,1.5,0.1,Iris-setosa
11 5.4,3.7,1.5,0.2,Iris-setosa
12 4.8,3.4,1.6,0.2,Iris-setosa
13 4.8,3.0,1.4,0.1,Iris-setosa
14 4.3,3.0,1.1,0.1,Iris-setosa
15 ...
16 ...
```

4. 代码实现

[project4.py](#), 使用[utils.py](#)

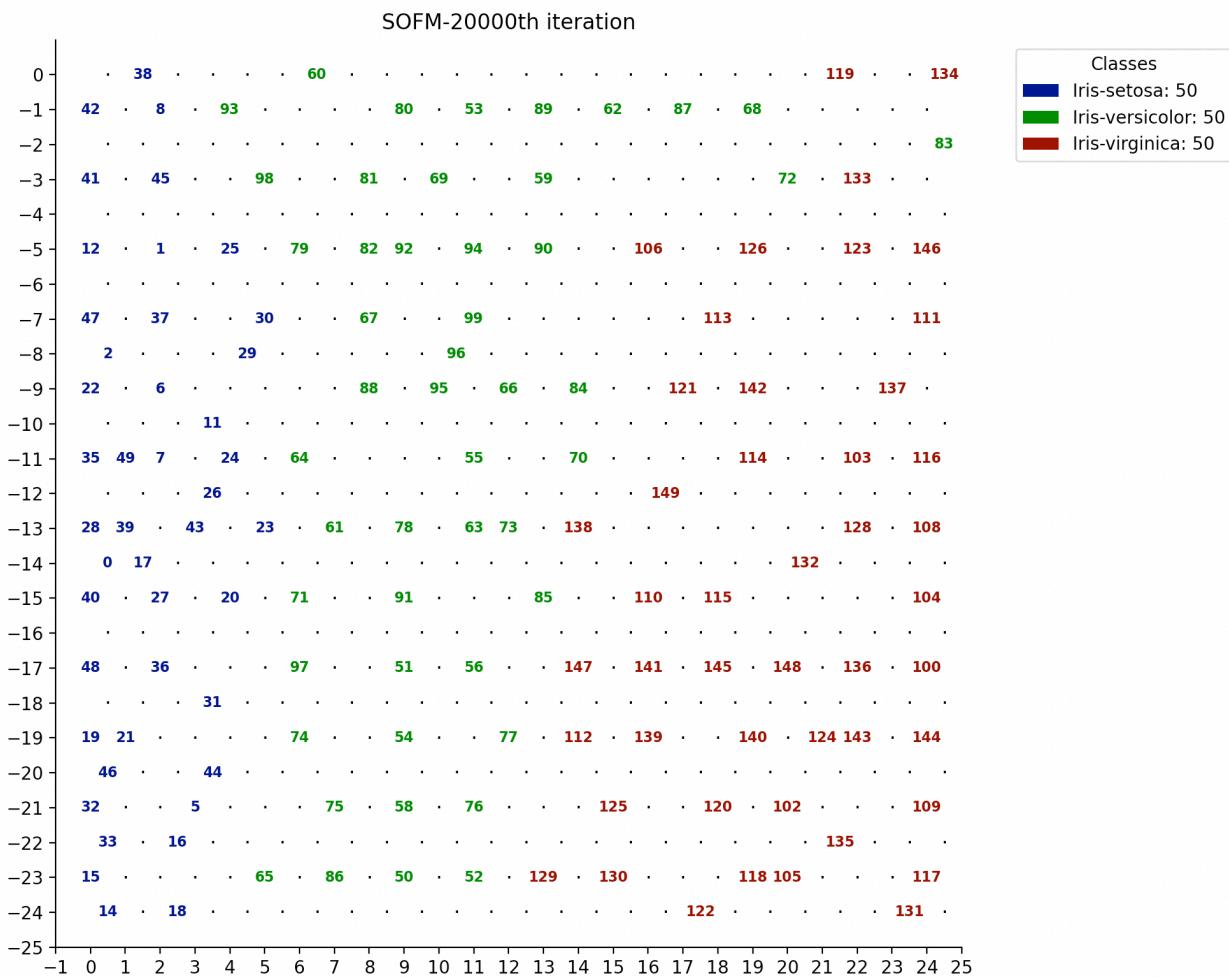
```
1 from utils import *
2
3 def main():
4     x, label0, label_dict, I, P, color_dict = InputPattern("./iris.data")
5     # print(color_dict)
6     use_classes_in_iris = True
7     use_wta_cluster = True if not use_classes_in_iris else False
8     if use_wta_cluster:
9         classes = np.loadtxt('./clusters_3.txt', dtype=str)
10        label_dict = {label0[i]: classes[i] for i in range(len(classes))}
```

```
11
12     unique_classes = np.unique(classes)
13     color_list = ['red', 'blue', 'green', 'purple', 'cyan', 'orange',
14     'magenta', 'lime', 'navy', 'darkred', 'darkgreen', 'gold', 'teal']
15     random.shuffle(color_list) # Randomly shuffle the color list
16     color_dict = {unique_classes[i]: color_list[i] for i in
17     range(len(unique_classes))}#
18     # print(label_dict)
19     print(color_dict)
20
21     # print(label_dict)
22     w = np.random.random((N, I))
23     w = SOFM(3000, 0.5, 0.04, 10, 1, w, x, I, P)
24     label = Calibration(w, x, label0, I, P)
25     print("\n\nResult after the first 1,000 iterations:\n")
26     # PrintResult(label)
27     # PrintResult_figure(label, label_dict, 3000, color_dict)
28     w = SOFM(17000, 0.04, 0.0, 1, 1, w, x, I, P)
29     label = Calibration(w, x, label0, I, P)
30     print("\n\nResult after 10,000 iterations:\n")
31     PrintResult(label)
32     PrintResult_figure(label, label_dict, 20000, color_dict)
33
34
35 if __name__ == "__main__":
    main()
```

5.结果讨论

1.Using given classes

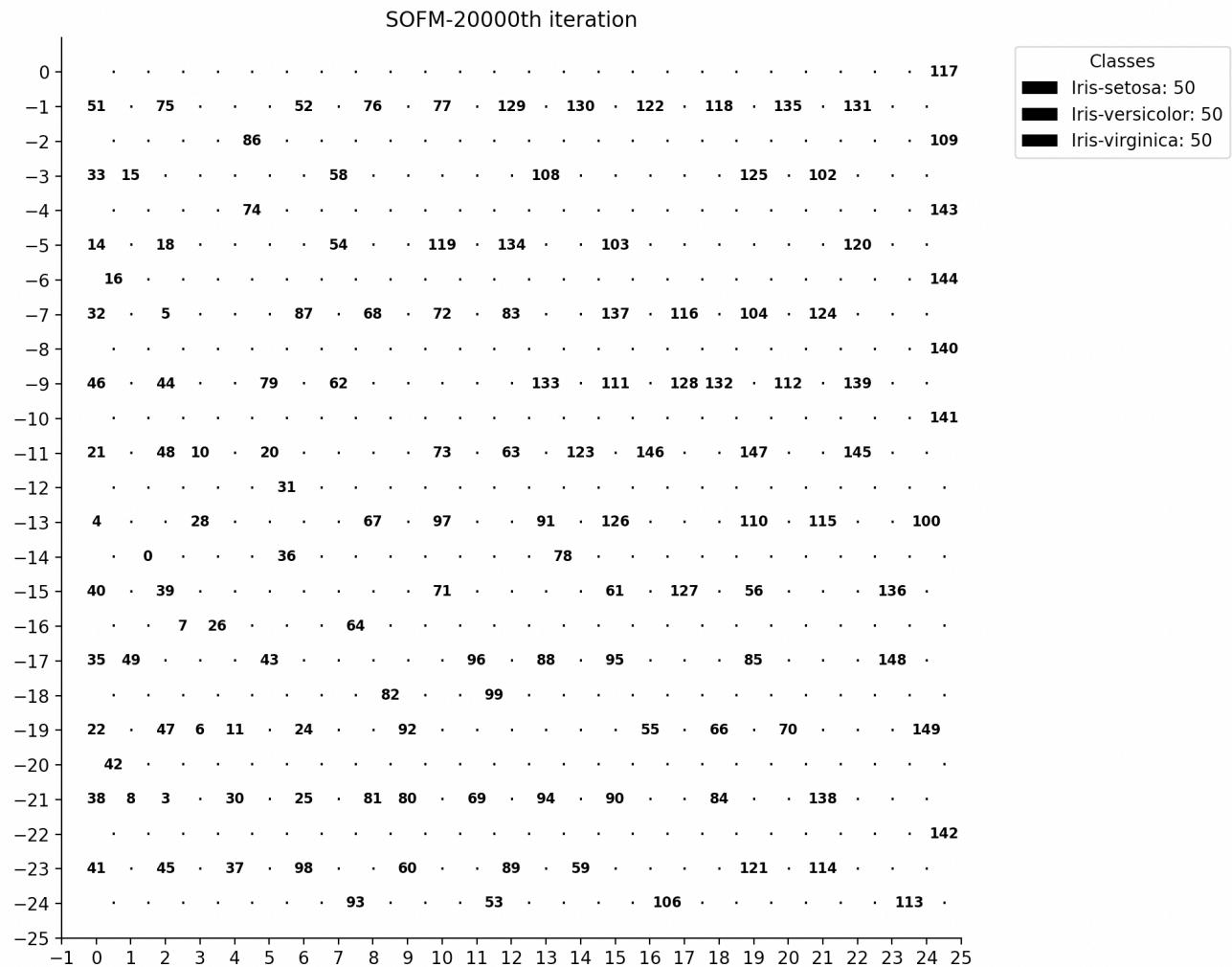
给Iris数据集150组数据分配0-149的标签，根据数据集自带的类别信息对每个标签进行分类，然后运用SOFM算法将高维数据映射到二维平面，每个类别分别着色



可以看出数据在降维后保留了一定的拓扑结构，同时数据的分布和数据集给定的分类相吻合，同一类的数据被准确聚类在了一起。

2. Don't use given class

SOFM的应用场景中，通常数据集本身是不具有分类信息的，运用算法后如图所示：



可以看出虽然SOFM压缩数据后保留了原始数据的拓扑结构，但是没法显示地将数据划分为不同的簇，这意味着我们无法从SOFM的结果中看出明确的聚类

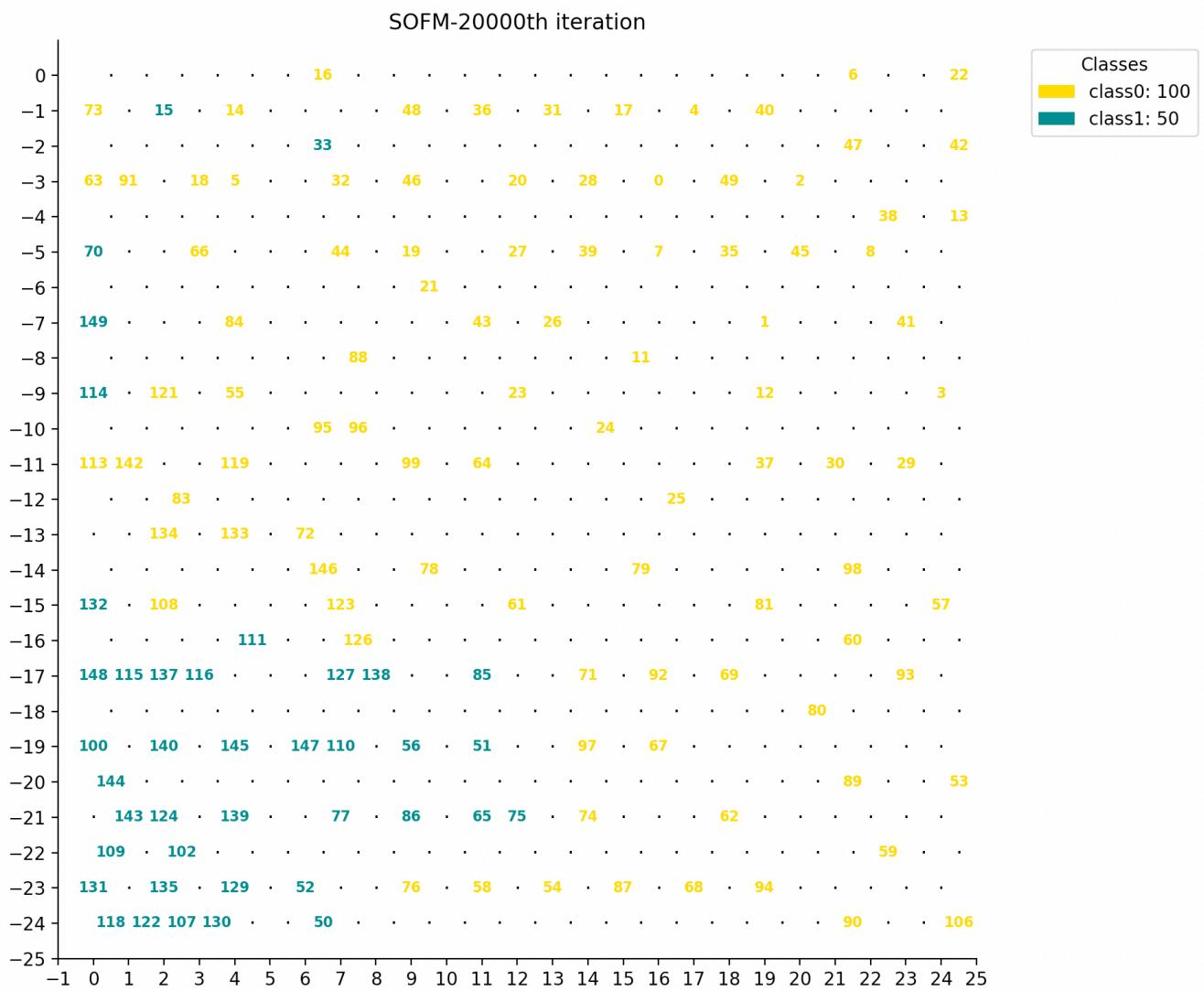
3.Cluster then SOFM

为了解决上面的问题，即既要压缩数据后保存拓扑结构，又要完成聚类，我们可以先使用WTA算法对数据集进行聚类，然后再用SOFM算法

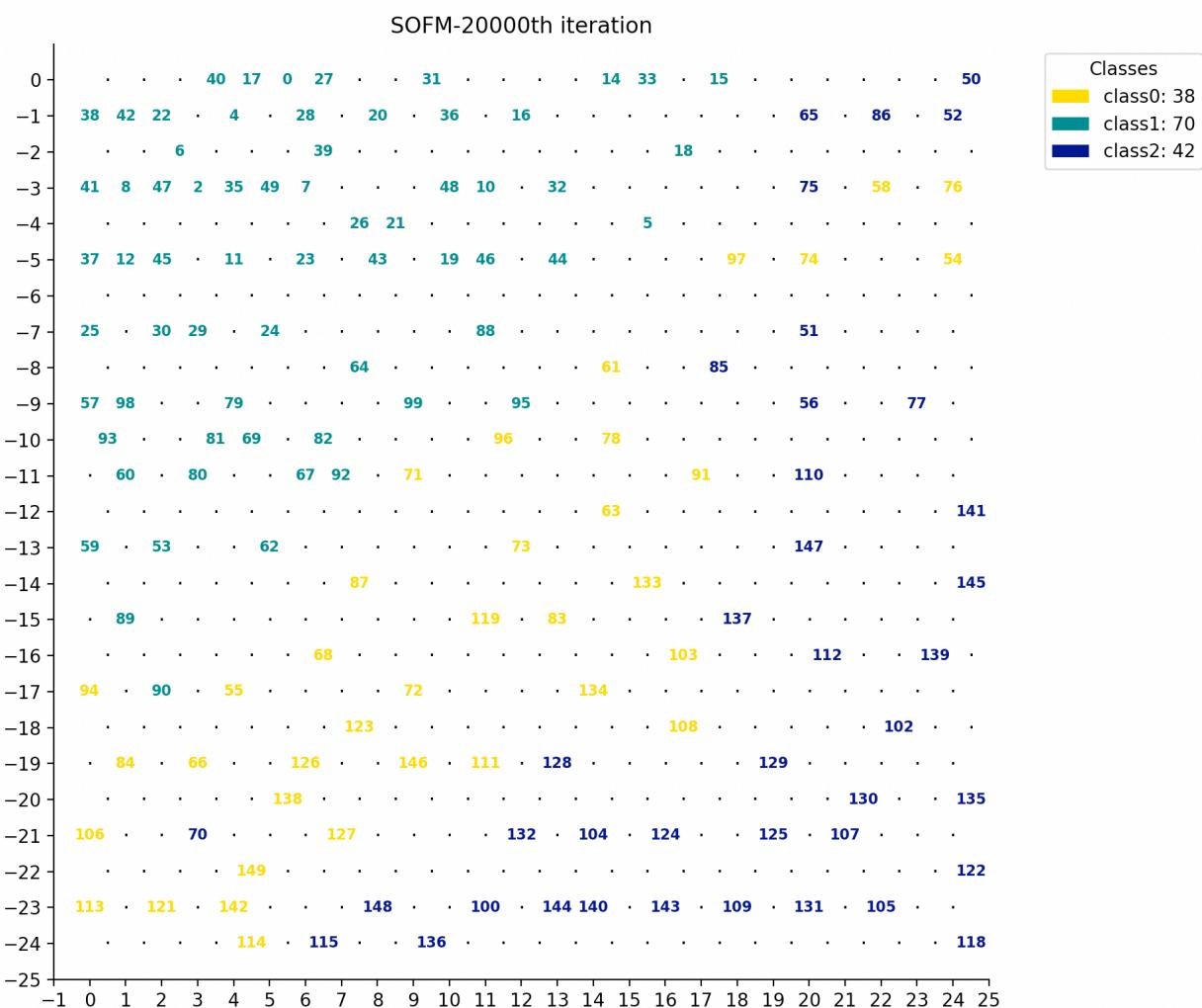
[output_cluster.py](#)可以自定义聚类的数量，同时将聚类结果输出到txt文件中

[project4.py](#)可以读取包含聚类信息的txt文件，既而完成SOFM算法

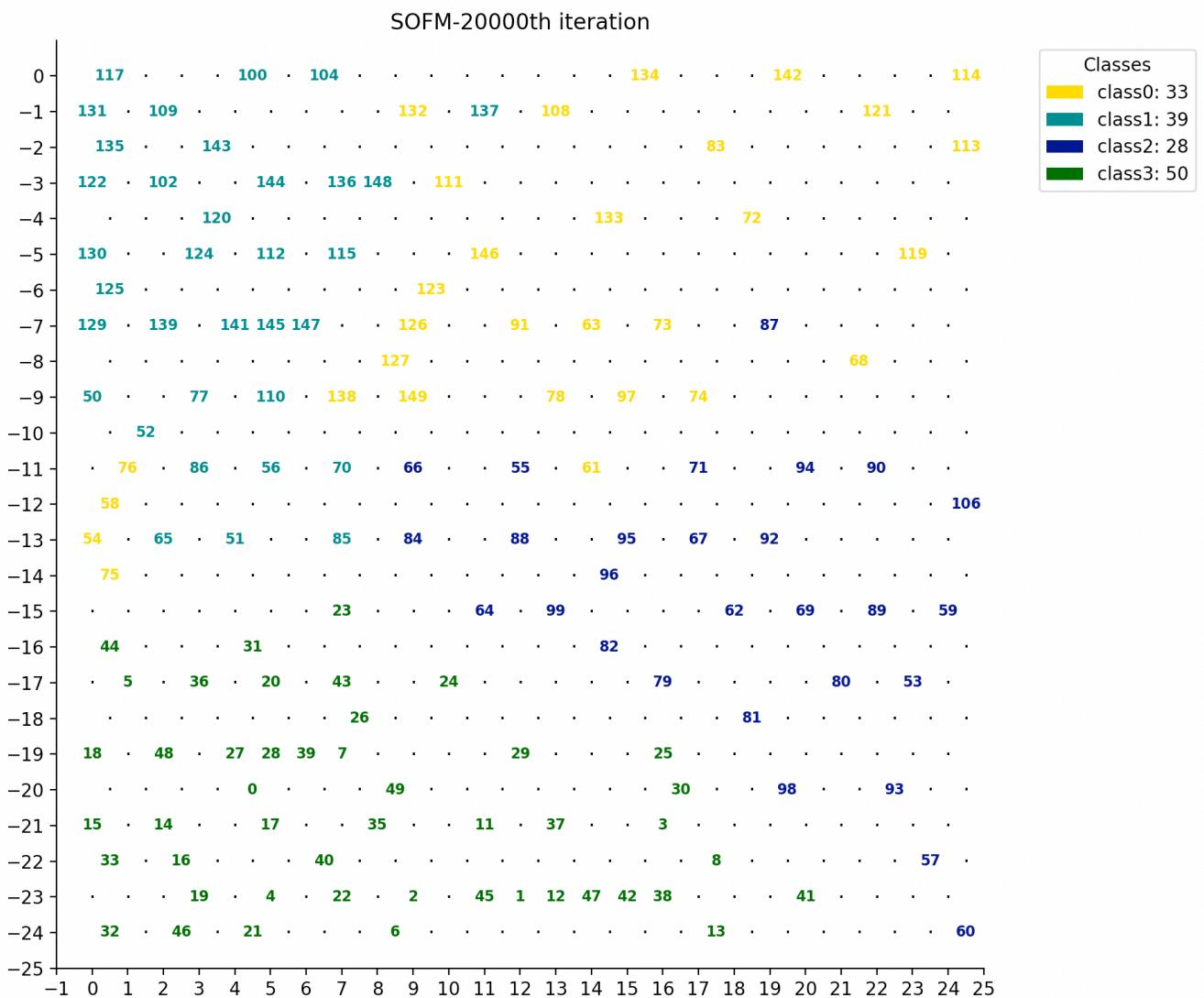
1.n_clusters = 2



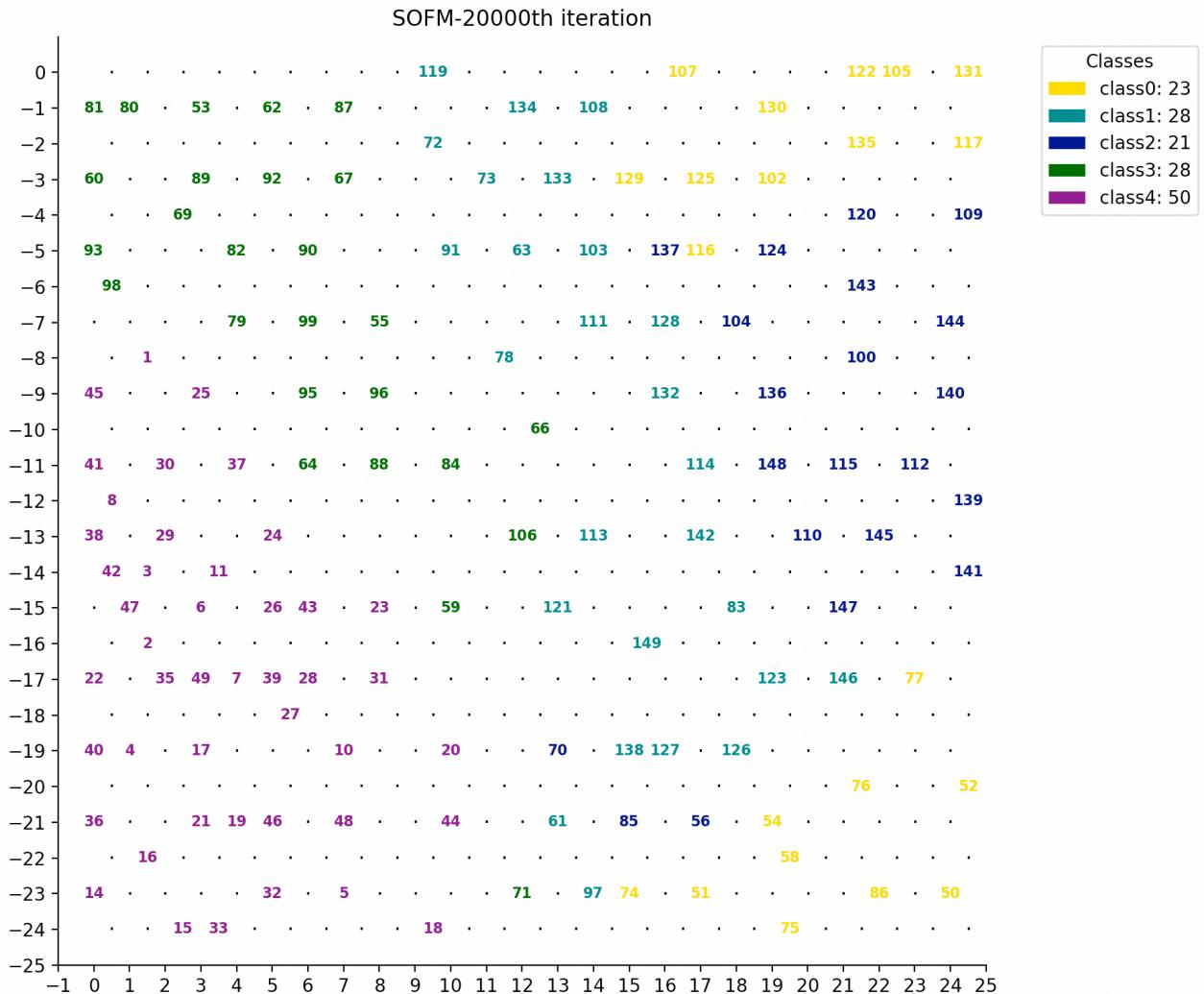
1.n_clusters = 3



3.n_clusters = 4



4.n_clusters = 5



可以看出，聚类的结果比较准确，同一类型的数据分布比较集中，同时也保留了拓扑结构。

6.Conclusion

上面的图像可以看出，WTA和SOFM算法结合有着很好的效果，既可以压缩数据的同时保留原始数据的拓扑结构，又可以准确的聚类。

同时通过观察图像，可以发现图像并不能完全显示所有的150个标签，原因是：

每一个输入模式都对应一个距离最近的神经元，因此存在一种可能，多个输入模式对应的是同一个神经元。

在迭代过程中，如果两个或更多的输入模式的“最接近”神经元相同，那么在我们的标签赋值过程中，后来的输入模式的标签就会覆盖先前的输入模式的标签，导致图像的显示中会漏掉一些标签。

相关代码如下：

```
1 def Calibration(w, x, label0, I, P):
2     label = ['.'] * N
3     for p in range(P):
4         d = np.linalg.norm(w - x[p], axis=1) # calculate the Euclidean
distance
5         n0 = np.argmin(d) # find the index of the smallest distance
6         label[n0] = str(label0[p])
7     return label
```

也就是说不同的p可能对应同一个n0