t-Distributed Stochastic Neighbor Embedding (t-SNE)

In this exercise, you will implement elements of the t-SNE algorithm described in the paper by Laurens van der Maaten (available on ISIS), and analyze its behavior. As a reminder, here are the main steps of the t-SNE procedure described in the paper:

- compute pairwise affinities $p_{j|i}$ with perplexity perp using $p_{j|i} = \frac{\exp(-||x_i - x_j||^2/2\sigma^2)}{\sum_{k \neq i} \exp(-||x_i - x_k||^2/2\sigma^2)}$
- Optimize the perplexity for each element i to give the target perplexity (provided in `utils.py`)
- Symmetrize the affinity matrix using $p_{ij} = \frac{p_{j|i} + p_{i|j}}{2N}$
- Consider an initial embedding Y^0
- Repeat for multiple iterations:
  - Compute the affinities in the embedded space $q_{ij} = \frac{(1+||y_i - y_j||^2)^{-1}}{\sum_{k \neq i}(1+||y_i - y_k||^2)^{-1}}$
  - Compute the gradient $\frac{\partial C}{\partial Y}$ using $\frac{\partial C}{\partial Y_i} = 4 \sum_j (p_{ij} - q_{ij})(y_i - y_j)(1 + ||y_i - y_j||^2)^{-1}$
  - Update the embedding using the update rule $Y^t = Y^{t-1} + \eta \frac{\partial C}{\partial Y} + \alpha(t)(Y^{t-1} - Y^{t-2})$, where Y^t is the value of Y at time t, where $Y^t = (0, 0, \ldots, 0)$ for $t < 0$ and where $\alpha(t) = 0.5$ at the beginning of the training procedure and 0.8 towards the end.
- Return the final embedding Y^T where T is the number of iterations

Implementing t-SNE (30P)

You are asked to implement several functions that are used by the t-SNE algorithm. Their specification is given below. In their current form, they simply call functions of the module `solutions`, which is not provided. Replace these calls by your own implementation of the functions. Remark that most of the time, we work with log-probabilities. It is more convenient and numerically stable when the probabilities need to be defined or normalized. (See for example the function `scipy.misc.logsumexp` for that purpose.)

In [39]:

```
import scipy,scipy.spatial
import numpy as np

def student(Y):
    # Calculate the join log-probabilities log(q_ij) defined above
    #
    # input:  Y    - An Nx2 array containing the embedding
    # return: logQ - An NxN array containing log(q_ij)

    N = Y.shape[0]
    D2 = scipy.spatial.distance.cdist(Y,Y,'sqeuclidean')
    D = (1 + D2) ** -1
    ulogQ = np.log(D)
    logQ = ulogQ - np.log(D.sum() - N)
    return logQ

def objective(logP,logQ):
    # Calculate the objective of t-SNE to minimize. The objective is the
    # KL divergence C = KL(P||Q)
    #
    # inputs: logP - An NxN array containing log(p_ij)
    #         logQ - An NxN array containing log(q_ij)
    # return: C    - The value of the objective

    C = np.sum(np.exp(logP) * (logP - logQ))
    return C
```

```
def gradient(logP,Y):
    # Computes the gradient as described above.
    #
    #inputs: logP  - An NxN array containing log(p_ij)
    #        Y     - An Nx2 array containing the embedding
    #return: gradY - the gradient of the objective with respect to Y

    N = Y.shape[0]
    gradY = np.zeros((N,2))
    logQ = student (Y)
    deltP = np.exp(logP) - np.exp(logQ)
    D2 = scipy.spatial.distance.cdist(Y,Y,'sqeuclidean')
    for i in range(N):
        deltY = Y[i,:] - Y
        for j in range (N):
            gradY[i,:] += 4 * deltP[i,j] * deltY[j,:] / (1 + D2[i,j])
    return gradY
```

The code below implements t-SNE algorithm. It takes as input some unsupervised dataset `X` (a `Nxd` array), and compute a two-dimensional embedding starting from an initial embedding `Y0` (a `Nx2` array). Various training parameters can be specified as optional parameters. The t-SNE algorithm makes use of the functions that are defined above.

In [38]:

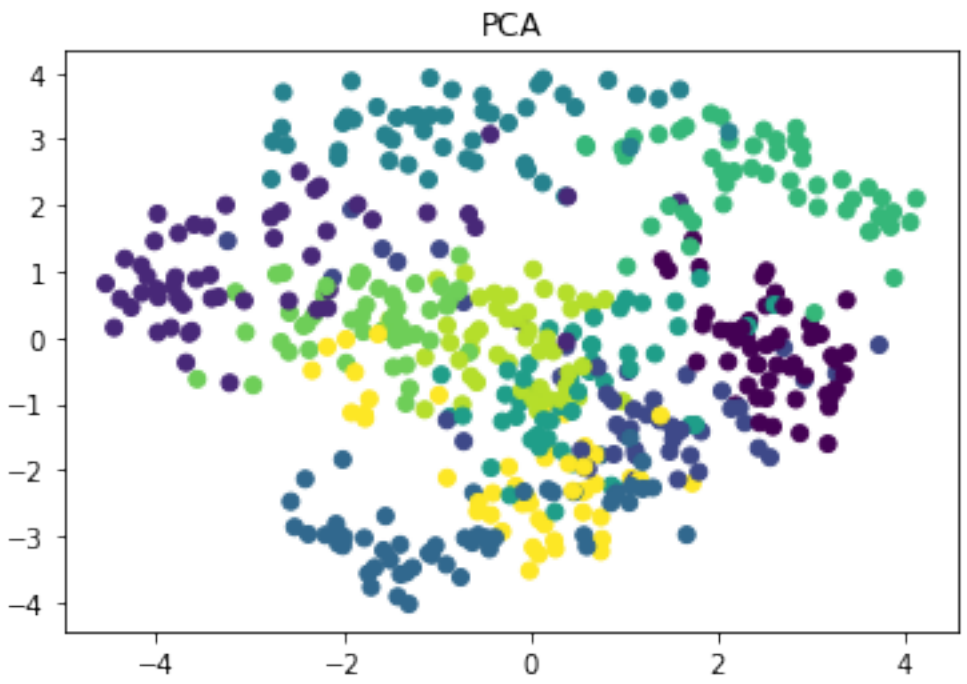
```
import utils
import numpy as np

def TSNE(X,Y0,perplexity=25,learningrate=1.0,nbiterations=250):

    N,d = X.shape

    print('get affinity matrix')

    # get the affinity matrix in the original space
    logP = utils.getaffinity(X,perplexity)

    # create initial embedding and update direction
    Y  = Y0*1
    dY = Y*0

    print('run t-SNE')

    for t in range(nbiterations):

        # compute the pairwise affinities in the embedding space
        logQ = student(Y)

        # monitor objective
        if t %50 == 0:
            print('%3d %.3f'%(t,objective(logP,logQ)))

        # update
        dY = (0.5 if t < 100 else 0.8)*dY + learningrate*gradient(logP,Y)
        Y = Y - dY

    return Y
```

We test the T-SNE algorithm on the handwritten digits dataset, and compare the found embedding with simple PCA analysis.

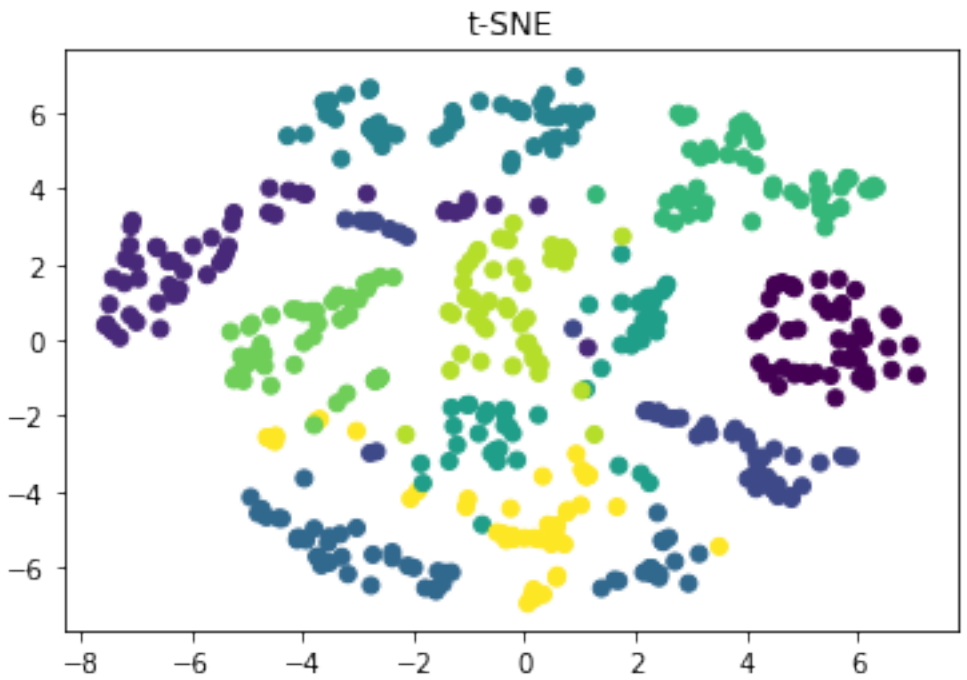
In [40]:

```
import utils
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
%matplotlib inline

# read input dataset
X,color=utils.get_data(mode=1)

# run PCA
U,W,_ = np.linalg.svd(X,full_matrices=False)
Y0 = U[:,:2]*W[:2]
plt.scatter(*Y0.T,c=color); plt.title('PCA')
plt.show()

# run TSNE starting with PCA embedding as an initial solution
Y = TSNE(X,Y0,perplexity=10,learningrate=5.0)
plt.scatter(*Y.T,c=color); plt.title('t-SNE')
plt.show()
```

```
Loading digits
```




```
get affinity matrix
 25 8.491 10.000
 50 10.445 10.000
 75 10.133 10.000
100 9.314 10.000
run t-SNE
  0 2.396
 50 2.180
100 2.041
150 1.817
200 1.660
```



Experimenting With t-SNE (20P)

The file `utils.py` contains a method `get_data(type)` that provides three datasets:

- A collection of digits (less complex than MNIST)
- Boston housing dataset
- Iris dataset

Using your implementation of t-SNE, and running it on the various dataset and with specific training parameters, answer the questions below. Along with your textual answers, include relevant results from running t-SNE in the code cell beneath each question where you should run code with certain parameters (perplexity, learning rate, choice of dataset) relevant to your answer.

How does perplexity and learning rate impact performance? What kind of extreme behaviour these parameters can cause?

We train dataset 3 using different parameters to illustrate the effect of perplexity and learning rate.

1. Perplexity

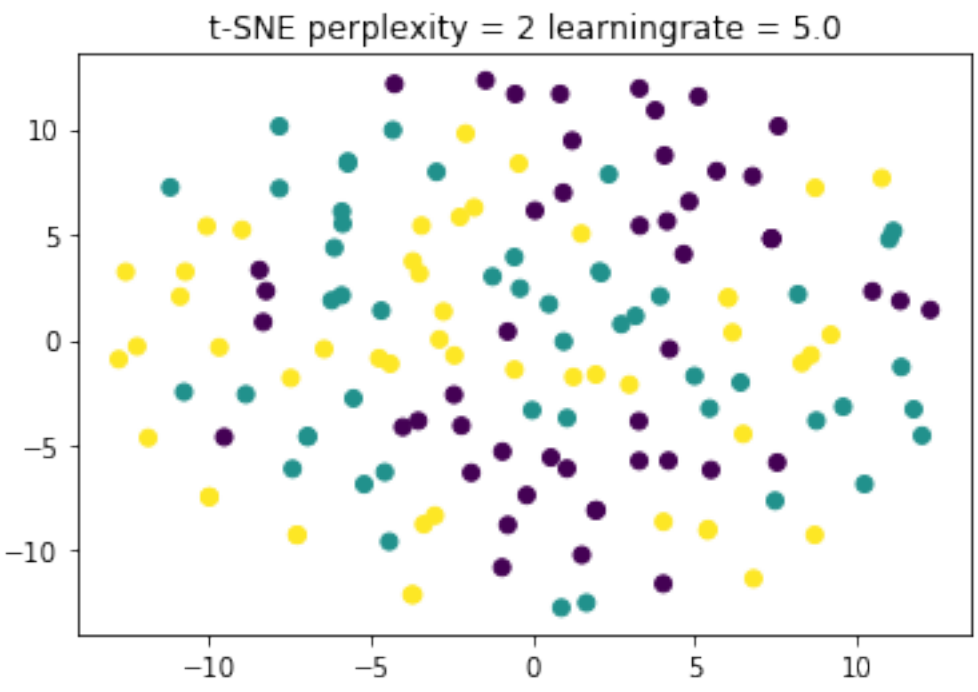
With perplexity values in the range (5 - 50) suggested by van der Maaten & Hinton, the diagrams do show these clusters, although with very different shapes. Outside that range, things get a little weird. With perplexity 2, local variations dominate. The image for perplexity 200, with merged clusters, illustrates a pitfall: for the algorithm to operate properly, the perplexity really should be smaller than the number of points. Implementations can give unexpected behavior otherwise.

1. Learning rate The higher learning rate makes the reduction of cost function faster, but too high makes it easy to miss the optimum. On the contrary, slow learning rate leads to slow convergence.

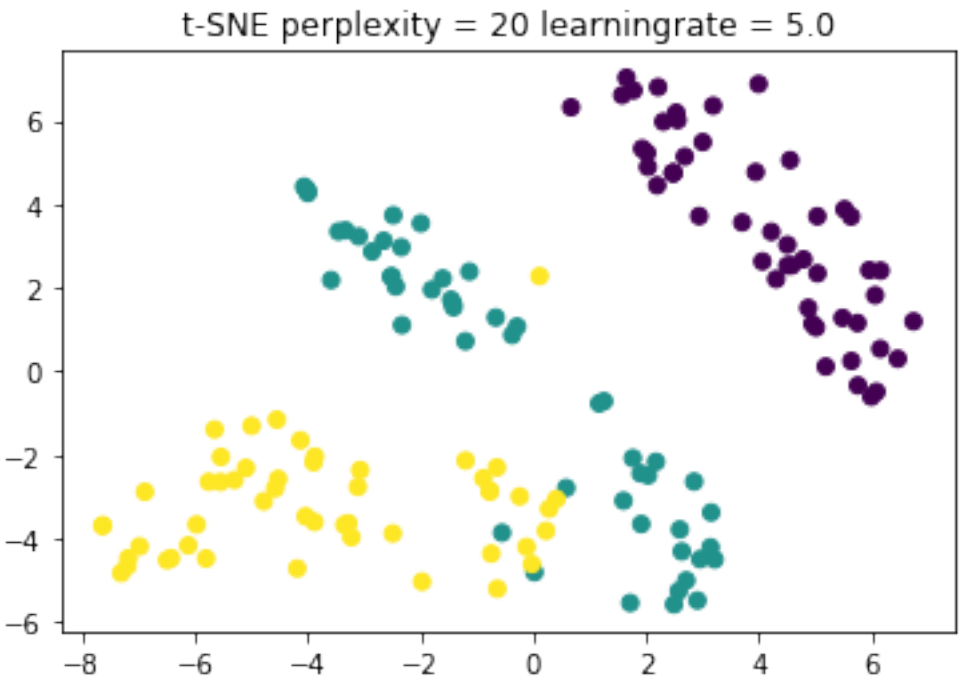
In [63]:

```
import utils
import numpy as np
import matplotlib
import matplotlib.pyplot as plt

### TODO: write your code here and run it

# read input dataset 3
X3,color = utils.get_data(mode=3)

# initialize using random numbers
Y0 = 0.001 * np.random.randn(150,2)

# test data using different perplexity
Y = TSNE(X3,Y0,perplexity=2,learningrate=5.0,nbiterations=250)
plt.scatter(*Y.T,c=color); plt.title('t-SNE perplexity = 2 learningrate = 5.0'
)
plt.show()

Y = TSNE(X3,Y0,perplexity=20,learningrate=5.0,nbiterations=250)
plt.scatter(*Y.T,c=color); plt.title('t-SNE perplexity = 20 learningrate = 5.0
')
plt.show()

Y = TSNE(X3,Y0,perplexity=200,learningrate=5.0,nbiterations=250)
plt.scatter(*Y.T,c=color); plt.title('t-SNE perplexity = 200 learningrate = 5.
0')
plt.show()

# test data using different learning rate
Y = TSNE(X3,Y0,perplexity=15,learningrate=1.0,nbiterations=250)
plt.scatter(*Y.T,c=color); plt.title('t-SNE perplexity = 15 learningrate = 1.0
')
plt.show()

Y = TSNE(X3,Y0,perplexity=15,learningrate=5.0,nbiterations=250)
plt.scatter(*Y.T,c=color); plt.title('t-SNE perplexity = 15 learningrate = 5.0
')
plt.show()

Y = TSNE(X3,Y0,perplexity=15,learningrate=25.0,nbiterations=250)
plt.scatter(*Y.T,c=color); plt.title('t-SNE perplexity = 15 learningrate = 25.
0')
plt.show()
```

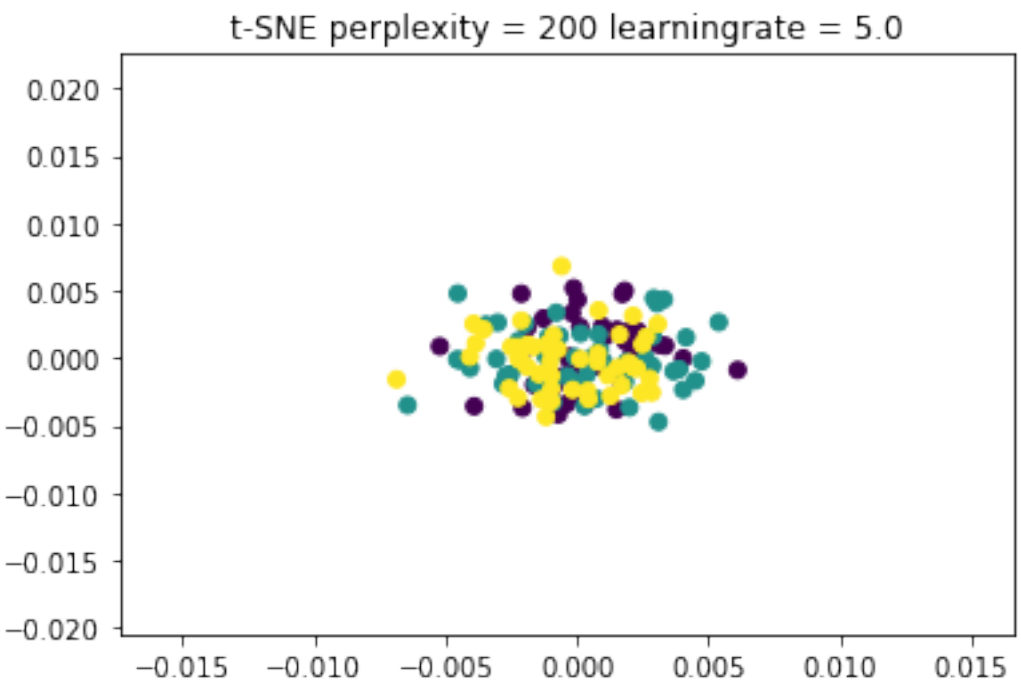
```
Loading iris
get affinity matrix
 25 1.981 2.000
 50 2.061 2.000
 75 2.041 2.000
100 2.009 2.000
run t-SNE
  0 4.276
 50 3.265
100 2.342
150 1.501
200 1.089
```



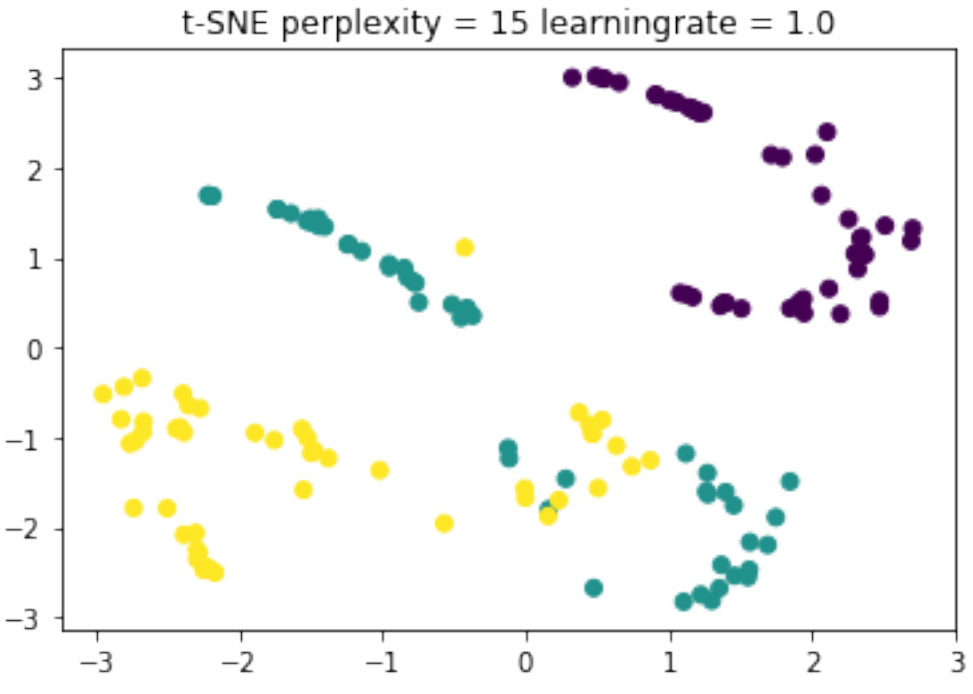

```
get affinity matrix
 25 19.904 20.000
 50 20.229 20.000
 75 20.205 20.000
100 19.961 20.000
run t-SNE
  0 1.940
 50 1.320
100 0.664
150 0.377
200 0.280
```



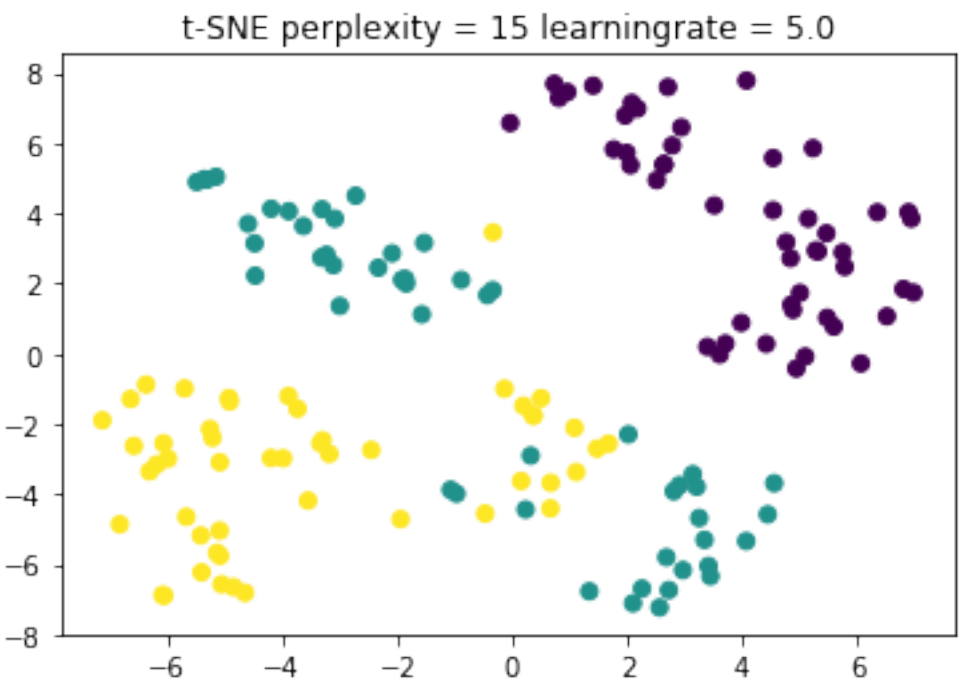
```
get affinity matrix
 25 149.998 200.000
 50 150.000 200.000
 75 150.000 200.000
100 150.000 200.000
run t-SNE
  0 -0.007
 50 -0.007
100 -0.007
150 -0.007
200 -0.007
```



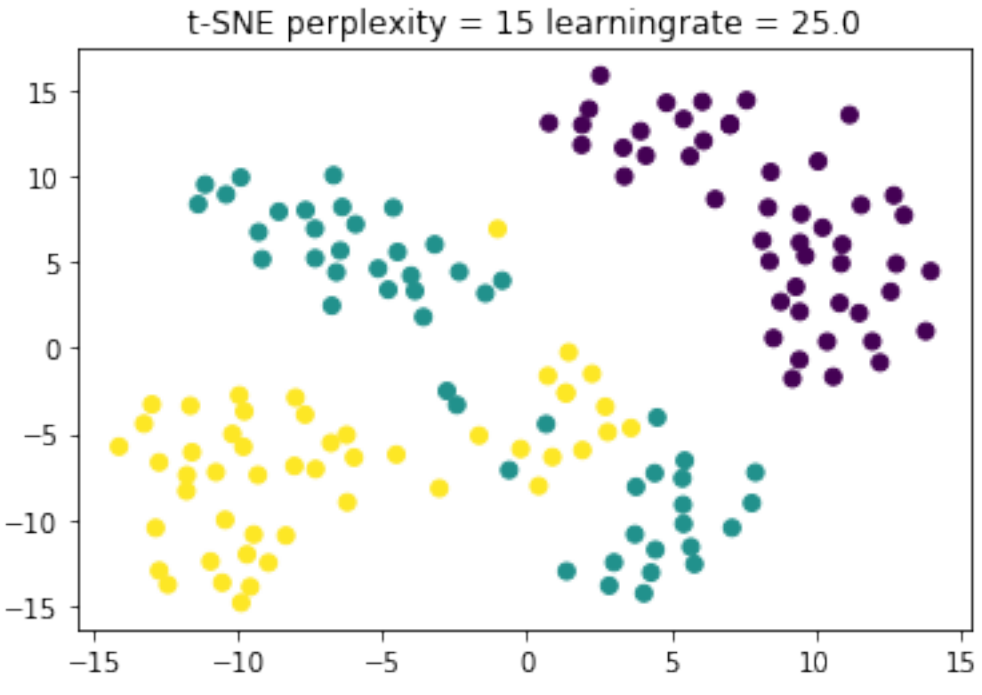

```
get affinity matrix
 25 14.757 15.000
 50 15.093 15.000
 75 15.042 15.000
100 14.998 15.000
run t-SNE
  0 2.225
 50 2.225
100 2.223
150 1.638
200 1.063
```



```
get affinity matrix
 25 14.757 15.000
 50 15.093 15.000
 75 15.042 15.000
100 14.998 15.000
run t-SNE
  0 2.225
 50 1.560
100 0.830
150 0.481
200 0.353
```




```
get affinity matrix
 25 14.757 15.000
 50 15.093 15.000
 75 15.042 15.000
100 14.998 15.000
run t-SNE
  0 2.225
 50 0.471
100 0.285
150 0.129
200 0.052
```



What kind of insight into the dataset you're dealing with can tSNE provide? Show one such example.

In [ ]:

```
### TODO: write your code here and run it
import utils
import numpy as np
import matplotlib
import matplotlib.pyplot as plt

### TODO: write your code here and run it

# read input dataset 2
X2,color = utils.get_data(mode=2)

# initialize using random numbers
Y0 = 0.001 * np.random.randn(500,2)

# test data using different perplexity
Y = TSNE(X2,Y0,perplexity=30,learningrate=10.0,nbiterations=250)
plt.scatter(*Y.T,c=color); plt.title('t-SNE')
plt.show()
```

```
Loading boston housing
get affinity matrix
 25 29.941 30.000
 50 30.149 30.000
 75 30.067 30.000
100 29.969 30.000
run t-SNE
  0 2.731
 50 2.706
100 1.782
150 1.053
```

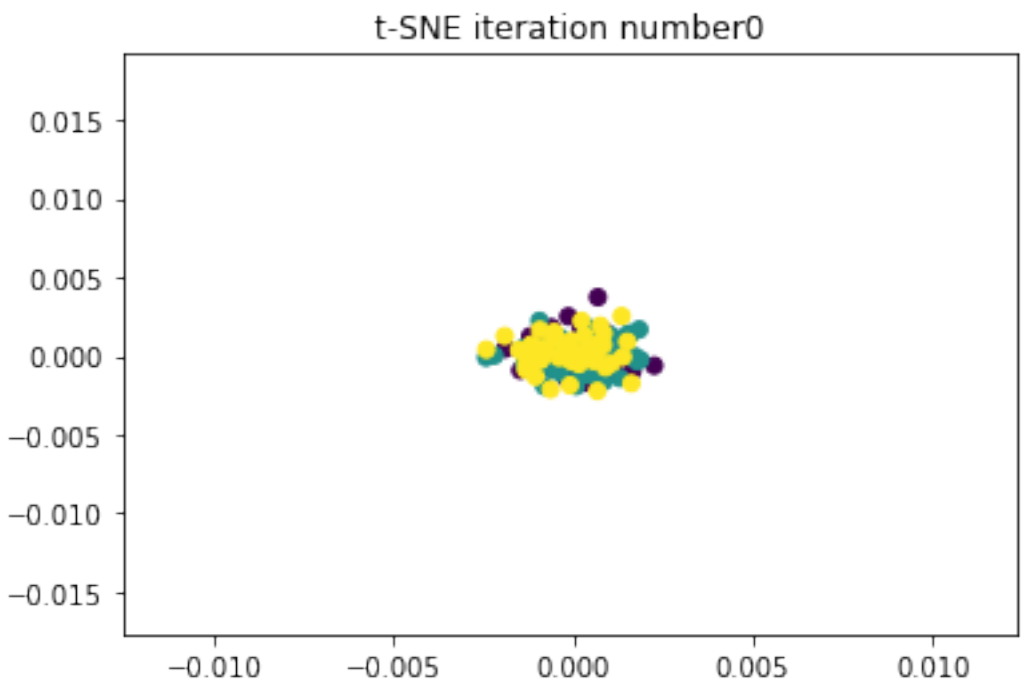
How does the embedding evolve during the optimization procedure (i.e. how are the clusters being formed progressively)?

Answer: We test data using dataset 3 with perplexity=15 and learning rate=5.0. A image is generated every 20 iterations. According to the results, the embedding start from a mess of random points. At the begining it evlove quickly into clusters （ in our case it's the first 50 iterations) and then finetuning(after 50 iterations),reaching a point of stability.

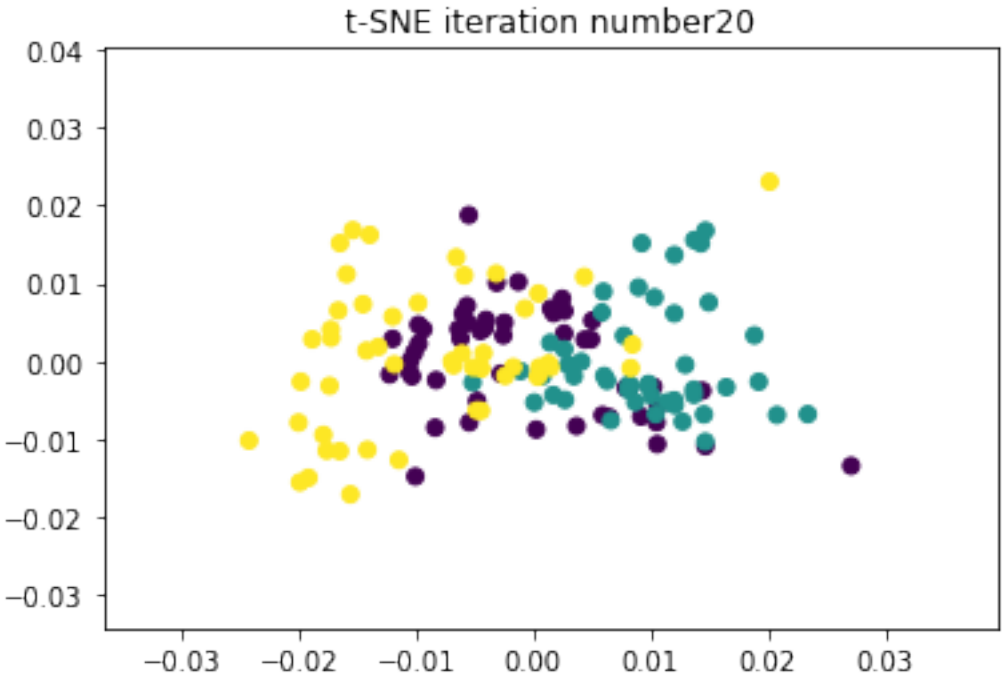
In [58]:

```
### TODO: write your code here and run it

# read input dataset 3
X3,color = utils.get_data(mode=3)

# initialize using random number
Y0 = 0.001 * np.random.randn(150,2)

# test data using dataset3 ,with parameters: perplexity=15 learning rate=5.0 .
for i in range(10):

    #plot the pattern every 20 iterations
    plt.scatter(*Y0.T,c=color); plt.title('t-SNE iteration number' + str(i*20)
)
    plt.show()

    Y0 = TSNE(X3,Y0,perplexity=15,learningrate=5.0,nbiterations=20)
```

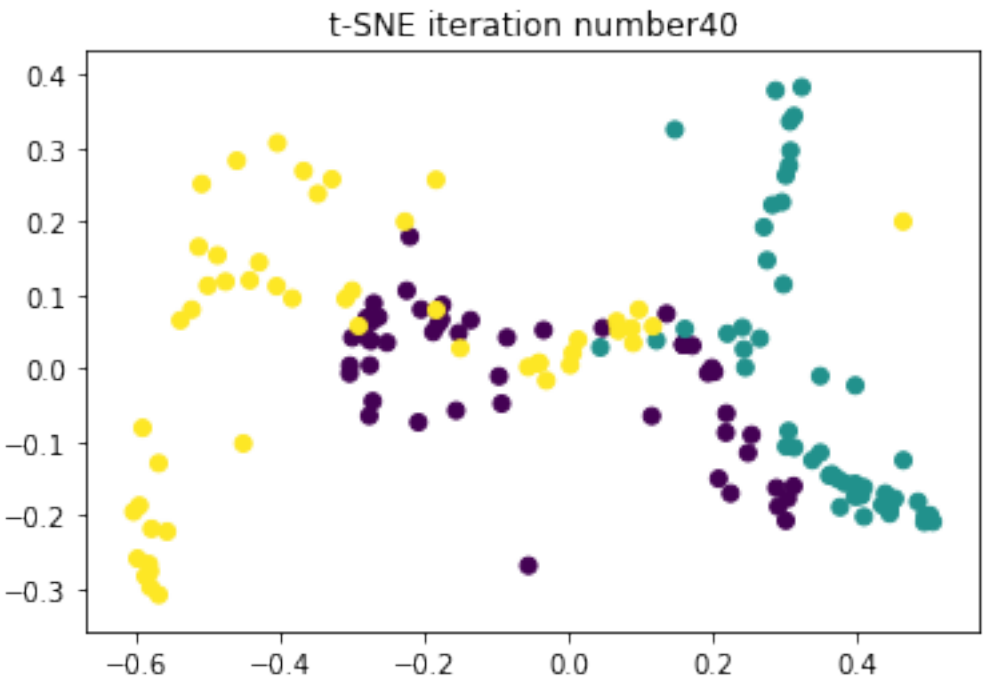
```
Loading iris
```



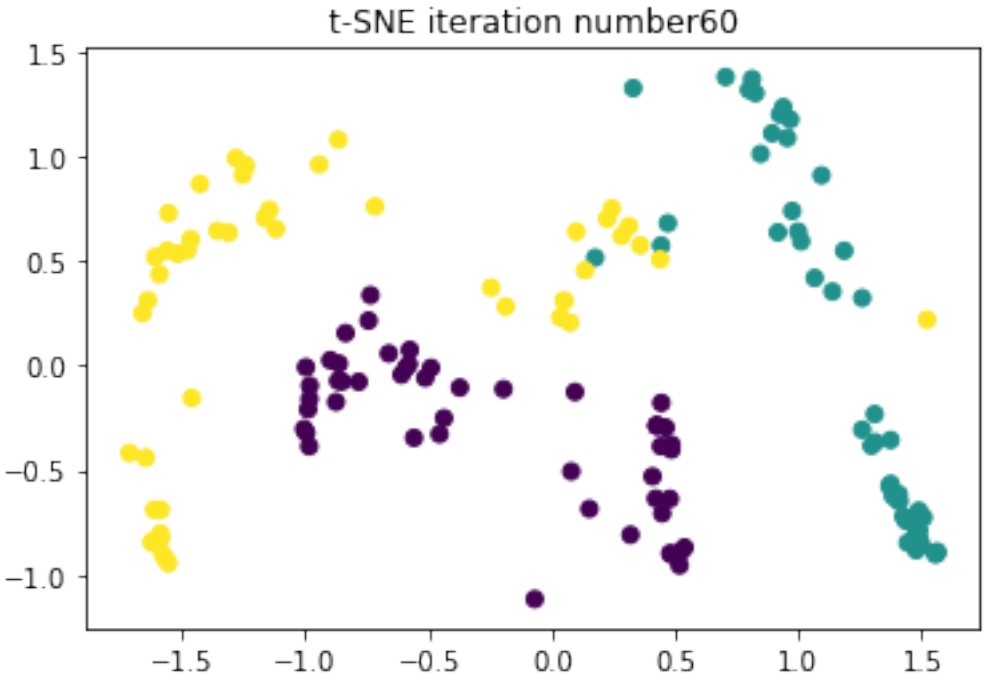

```
get affinity matrix
 25 14.757 15.000
 50 15.093 15.000
 75 15.042 15.000
100 14.998 15.000
run t-SNE
  0 2.225
```



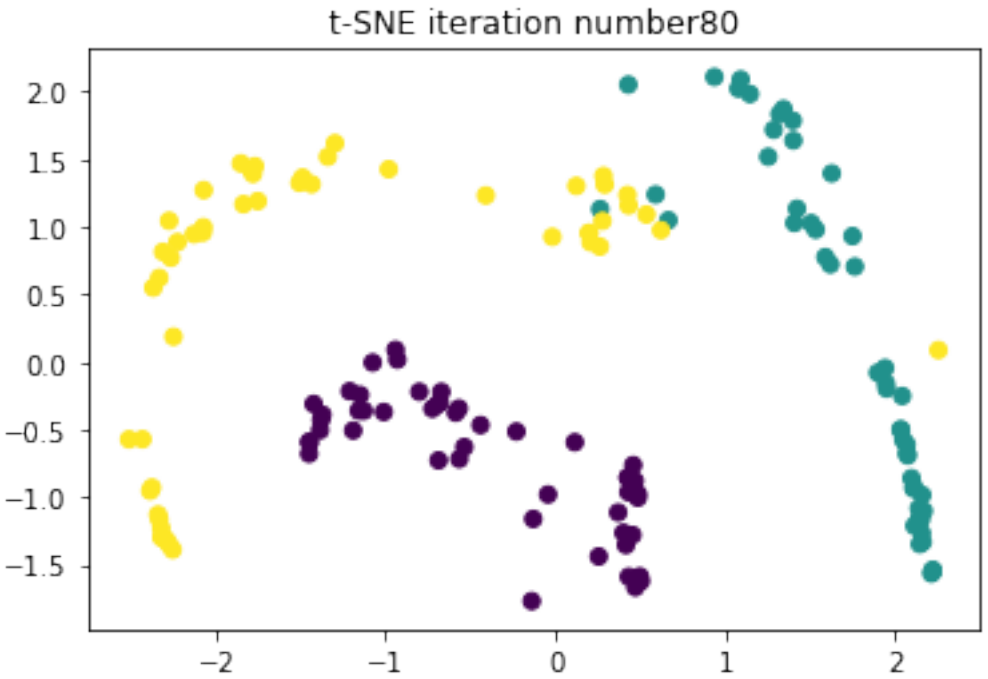

```
get affinity matrix
 25 14.757 15.000
 50 15.093 15.000
 75 15.042 15.000
100 14.998 15.000
run t-SNE
  0 2.224
```



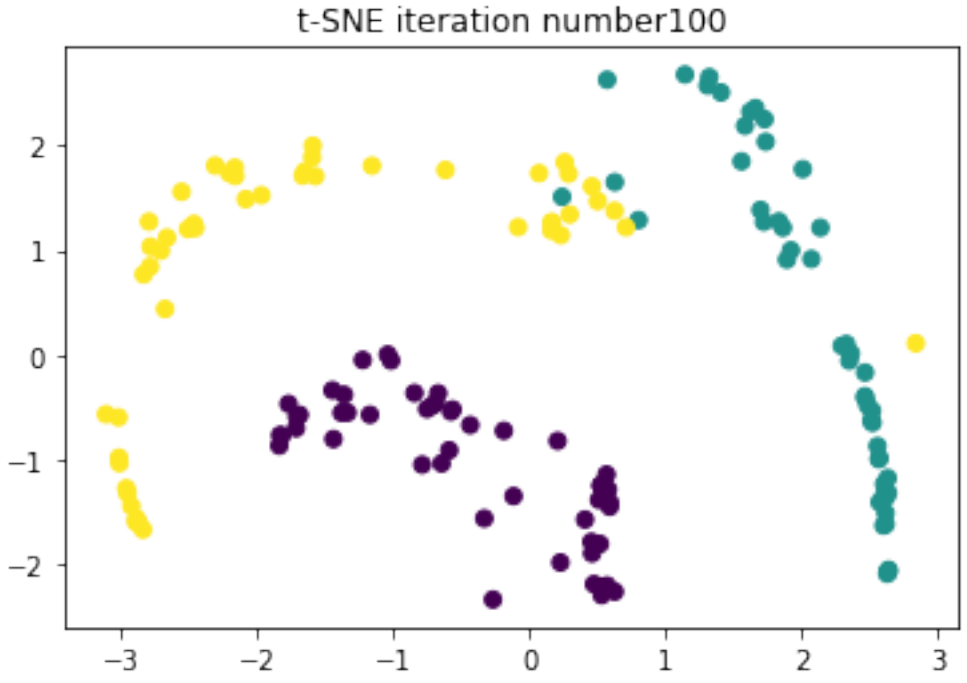

```
get affinity matrix
 25 14.757 15.000
 50 15.093 15.000
 75 15.042 15.000
100 14.998 15.000
run t-SNE
  0 2.037
```



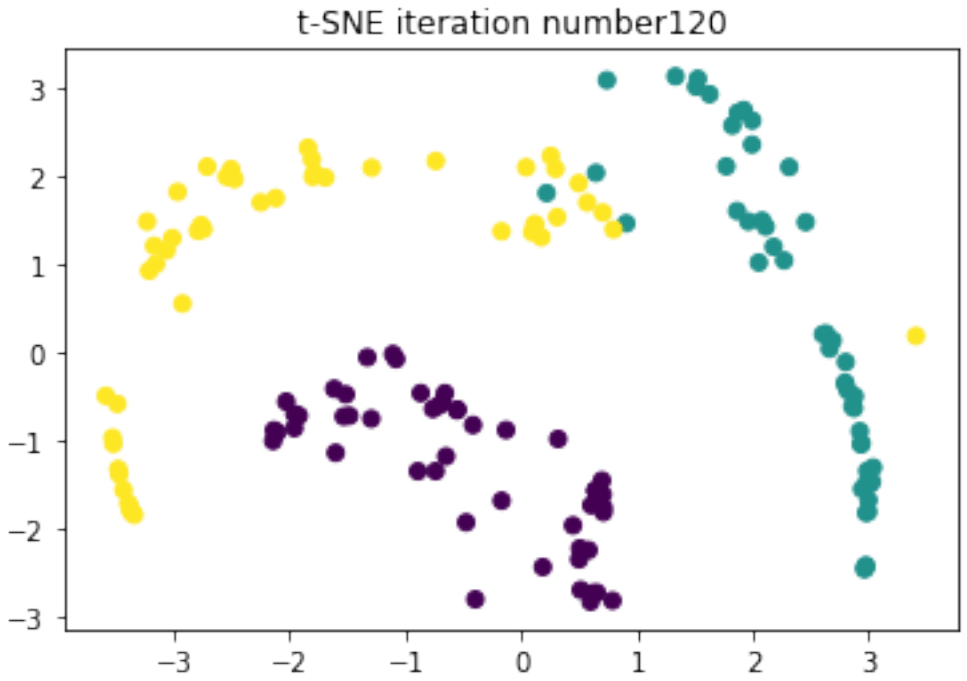

```
get affinity matrix
 25 14.757 15.000
 50 15.093 15.000
 75 15.042 15.000
100 14.998 15.000
run t-SNE
  0 1.371
```



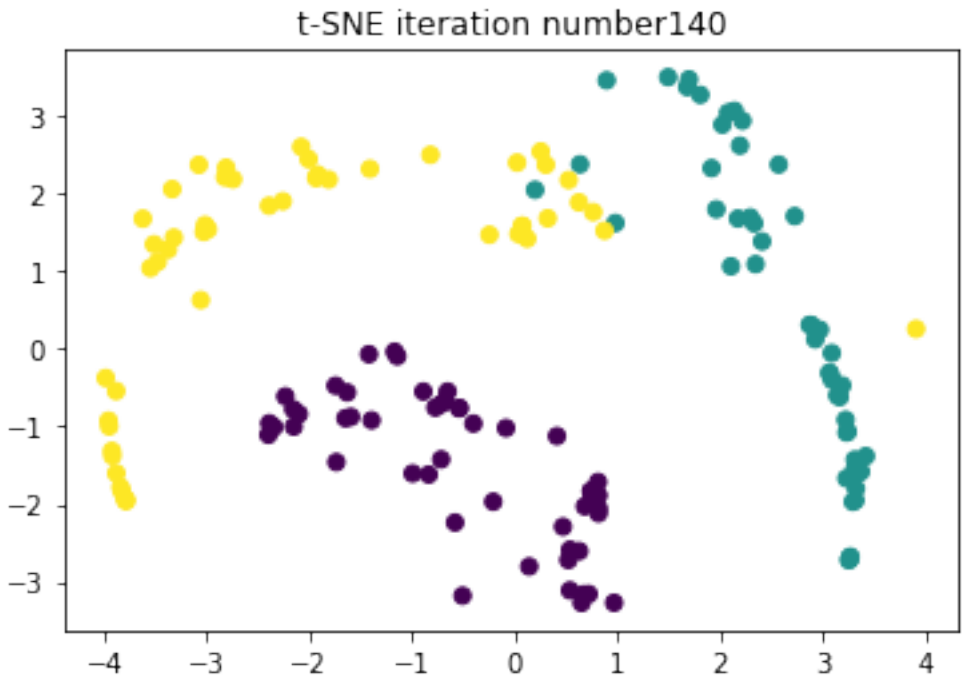

```
get affinity matrix
 25 14.757 15.000
 50 15.093 15.000
 75 15.042 15.000
100 14.998 15.000
run t-SNE
  0 1.028
```



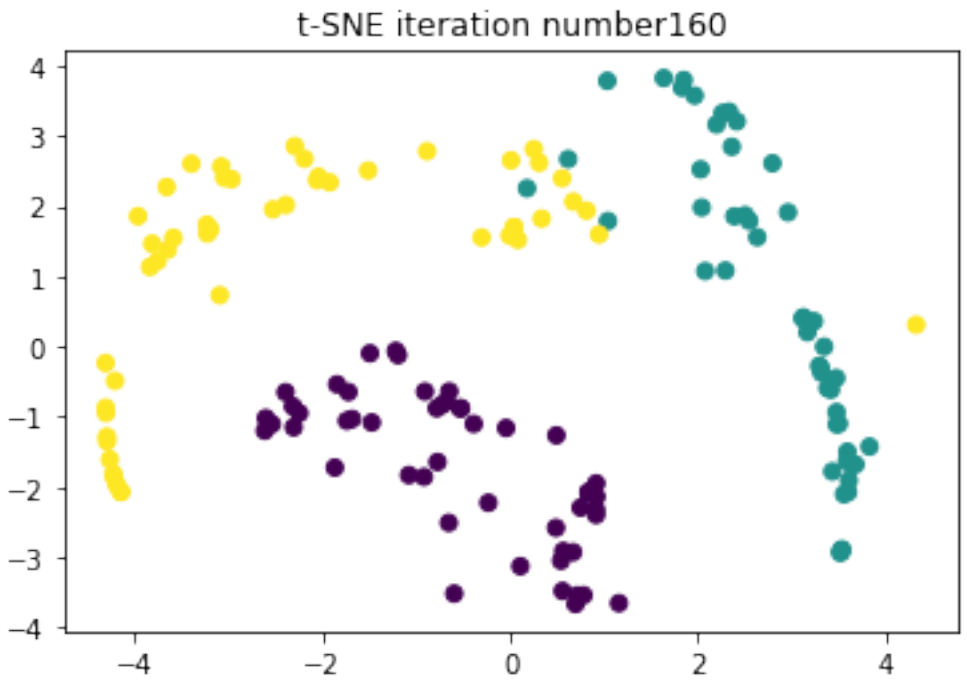

```
get affinity matrix
 25 14.757 15.000
 50 15.093 15.000
 75 15.042 15.000
100 14.998 15.000
run t-SNE
  0 0.858
```



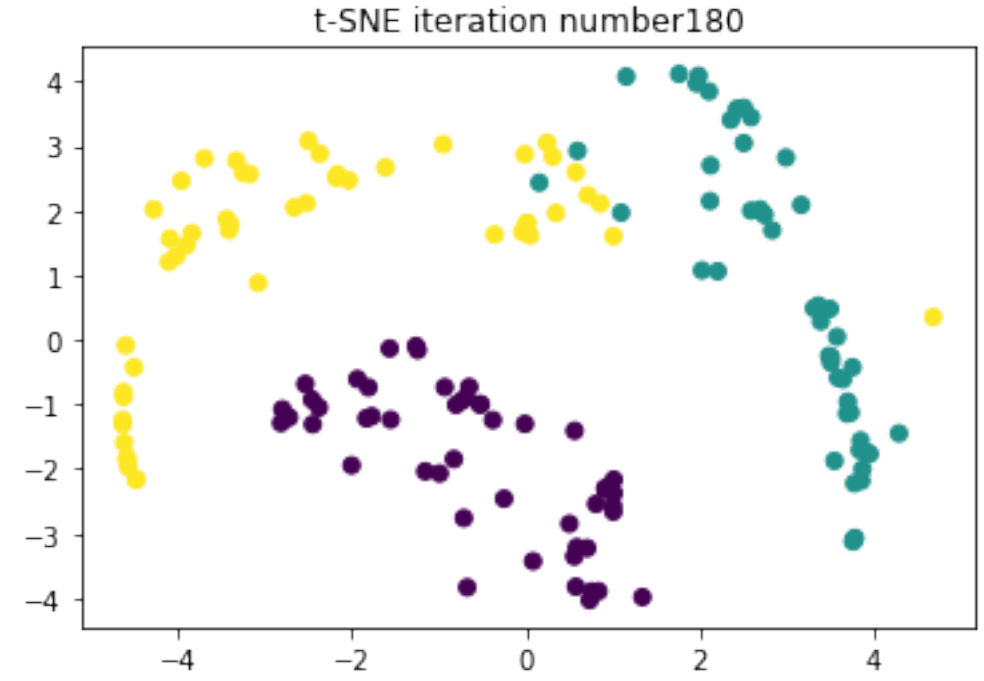

```
get affinity matrix
 25 14.757 15.000
 50 15.093 15.000
 75 15.042 15.000
100 14.998 15.000
run t-SNE
  0 0.751
```




```
get affinity matrix
 25 14.757 15.000
 50 15.093 15.000
 75 15.042 15.000
100 14.998 15.000
run t-SNE
  0 0.673
```




```
get affinity matrix
 25 14.757 15.000
 50 15.093 15.000
 75 15.042 15.000
100 14.998 15.000
run t-SNE
  0 0.613
```




```
get affinity matrix
 25 14.757 15.000
 50 15.093 15.000
 75 15.042 15.000
100 14.998 15.000
run t-SNE
  0 0.564
```