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 $Y^t = (0, 0, ..., 0)$ 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 [1]: import solutions

        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)

            logQ = solutions.student(Y)
            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 = solutions.objective(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

            gradY = solutions.gradient(logP,Y)
            return gradY
```

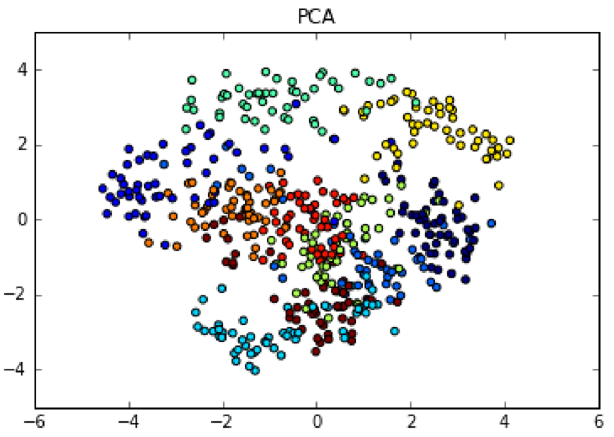
The code below implements t-SNE algorithm. It takes as input some unsupervised dataset X (a Nxd array), and compute a two-dimensional embedding starting from an initial embedding Y0 (a Nx2 array). Various training parameters can be specified as optional parameters. The t-SNE algorithm makes use of the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

```
In [2]: 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
```

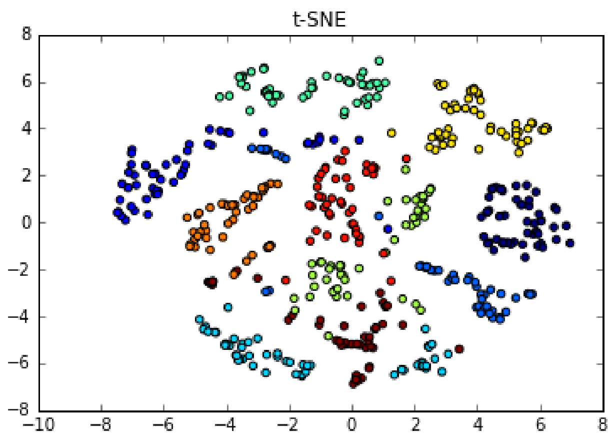
We test the T-SNE algorithm on the handwritten digits dataset, and compare the found embedding with simple PCA analysis.

```
In [3]: 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 2.409
 50 2.197
100 2.062
150 1.845
200 1.695
```



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?

[TODO: write your answer here]

```
In [4]: ### TODO: write your code here and run it
```

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

[TODO: write your answer here]

```
In [5]: ### TODO: write your code here and run it
```

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

[TODO: write your answer here]

```
In [6]: ### TODO: write your code here and run it
```