Image Interpretation – Assignment 6

This assignment on **Unsupervised Learning** covers the labs from the 28th of November and 5th of December and is graded by a maximum of 15 points.

In order to submit the results, send a ZIP file with your implemented code (functions prefixed with `ii_` and any "helper" functions you wrote) to

`<riccardo.delutio@geod.baug.ethz.ch>`

with subject

`Image Interpretation 2019 Assignment 6`

no later than on the

`12th of December, 2019.`

Your functions should work when called by the provided test code (functions prefixed with `test_`) which **must not** be modified. When run, they should produce a plausible output, no warnings, and no unnecessary output (remember to end your statements with semicolons). In order to help you verify whether or not your code is doing what it is supposed to, reference images `ref_`-images that show example output for each test script come with the provided code.

In addition to the functions, include a report (**max. 8 pages** PDF) explaining the structure of the code and the `python`/MATLAB functions used. This includes the reasons for choosing particular functions as well as a short justification of their parameter setting. For the more complicated tasks, the choice of the underlying data structures and algorithms should be explained too. We encourage you to add also diagrams, illustrations, and figures into the report when appropriate, but it is not necessary to copy the related theory from the lecture slides. The report should not contain any code snippets (but the code should contain comments if appropriate).

Team work is not allowed. Everybody implements his/her own code and writes his/her own report. Discussing issues with others is fine, sharing code and/or report with others is **not**. If you use any code fragments found on the Internet, make sure you reference them properly.

As opposed to the last assignment this one does not rely on built-in functions. You are asked to implement all functions from scratch completely on your own. This is done for two reasons: (i) k-means and mean shift are two very important techniques in pattern recognition that anyone taking such a course should have implemented once to fully understand them and (ii) implementing them is feasible and straightforward. All equations that have to be implemented are given and each task description takes you step-by-step through the code.

k-means (exercise 1) and mean shift (exercise 3) should both be run with both given test scripts. Inside the first test script the simulated data stays fixed for each trial whereas a new random sample is drawn if running the second test script more than once. This is to show you what impact different parameter settings have (the first test script) and what impact different data distributions/patterns have if parameters remain fixed (the second test script).

Exercise 1. 5 P.

The k-means clustering is a standard approach for clustering data. Due to its simplicity it is a common tool in practice today. Implement the k-means algorithm in a function called `ii_kmeans` that should take 3 arguments:

1. input data: sample points in a N-by-2 matrix (number of rows is the number of samples, dimensionality of the input data will always be 2 for this exercise)
2. the number of clusters K, a scalar
3. the number of iterations, a scalar

It should return two values.

1. cluster indexes: a column vector with N rows, specifying the cluster index for each sample
2. cluster centers: a K-by-2 matrix, returning the cluster centers for each cluster

First, initialize the cluster centers by choosing K distinct points randomly from the input data. The MATLAB function `randsample` can be useful here. The next two steps are then repeated until the cluster assignments do not change any more or the number of iterations is reached (whichever comes first).

In the **assignment step**, determine the closest cluster center for each point and store its index in a vector of indexes. If no index changed during the assignment step, stop the iteration. In the **update step**, replace each cluster center with the mean of the points in the respective cluster. Note that it can happen that a cluster ends up without any points. In this case, you cannot compute the mean and should thus pick a random point from the input data and use its position as the new cluster center. After both steps (the **assignment step** and the **update step**), call the function `visualize_kmeans` which takes 4 arguments: input data, the vector of cluster indexes, cluster centers, and a scalar (set it to `false` for the call after the assignment step and to `true` for the call after the update step). This helps visualize what the algorithm is doing. Once running a test script click on the figure to proceed to the next step.

Report: Run your code with the first test script `test_kmeans1` (fixed data sample) and evaluate several different parameter settings. Next, run your code with the second test script `test_kmeans2` (newly random data sample per run) and keep parameters fixed for several runs. Then repeat with a different parameter setting and so on. Discuss all differences and results (and their theoretical reasons) in the report. Also think about what may cause k-means to fail (or at least underperform) and how one might reduce the risk of it happening.

Exercise 2. 5 P.

Kernel density estimation is an essential part of many state-of-the-art pattern recognition methods like mean shift, for example. Write a function `ii_kde` that estimates kernel densities, which takes four input arguments:

1. input data: sample points in a N-by-d matrix (number of rows is the number of samples, number of columns is the dimensionality of the feature space)
2. kernel bandwidth: the scale (width, size) of the kernel, a scalar
3. kernel type: which kernel to use, a string – `'parzen'`, `'normal'`, or `'epanechnikov'`. To compare strings you may find function `strcmpi` helpful. Note that in order to structure your code well it might be helpful to write an own function per kernel type that is called inside `ii_kde`.
4. query points: the points where the kernel density is to be estimated, a M-by-d matrix (M is the number of query points)

The function should return a column vector with M rows, where the i-th value of the vector is the kernel density at the position specified by the i-th row of the query point matrix.

Instead of the formulas from the lecture slides, use the following formulas to compute the kernel density estimate per query point:

$$KDE_{\text{Parzen}}(\mathbf{q}) = \frac{1}{h^d N} \sum_{i=1}^{N} \mathbb{1}\left(\|\mathbf{q} - \mathbf{x}_i\|_\infty \le \frac{h}{2}\right) \tag{1}$$

where $\|\mathbf{v}\|_\infty$ is the infinity norm, equal to the largest absolute value of the vector components, h is the kernel bandwidth, $\mathbf{q}$ is the center of the kernel (the query points), $\mathbf{x}_i$ is the current data point in feature space, and d is the dimension of the feature space (e.g. 1 if run with test script `test_kde_1d`). $\mathbb{1}(\cdot)$ is an indicator function, i.e. it is 1 if the argument is true and 0 otherwise. More precisely, the sum in Eq. 1 counts the number of input points that are located inside the kernel which is mathematically expressed via $\|\mathbf{q}-\mathbf{x}_i\|_\infty \leq \frac{h}{2}$.

While the `parzen` function counts all points that fall into a kernel with equal weight 1, `normal` and `epanechnikov` kernels apply weights as a function of the distance from the kernel's center point $\mathbf{q}$:

$$KDE_{\text{Normal}}(\mathbf{q}) = \frac{1}{h^d N}\frac{1}{(2\pi)^{\frac{d}{2}}}\sum_{i=1}^{N}\exp\left(-\frac{1}{2}\frac{\|\mathbf{q}-\mathbf{x}_i\|^2}{h^2}\right) \tag{2}$$

Here, $\|v\|$ is the usual 2-norm (or euclidean norm, i.e. the euclidean distance in feature space).

$$KDE_{\text{Epanechnikov}}(\mathbf{q}) = \frac{1}{h^d N}\frac{d+2}{2V_d}\sum_{i=1}^{N}\max\left(0, 1-\frac{\|\mathbf{q}-\mathbf{x}_i\|^2}{h^2}\right) \tag{3}$$

where V_d is the volume of the d-dimensional unit sphere (a sphere with radius 1).

$$V_d = \frac{\pi^{\frac{d}{2}}}{\Gamma(\frac{d}{2}+1)} \tag{4}$$

$\Gamma(\cdot)$ is the gamma function. It can be computed in MATLAB with the function `gamma`. The purpose of the max is to replace negative values with 0.

Report: In order to verify if your kernels are implemented correctly you may want to start with test script `test_kde_1d_onepoint`, which returns the kernel shapes (check against `ref_test_kde_1d_onepoint.png` or images on the corresponding lecture slide). Next, run `test_kde_1d` and `test_kde_2d` to compare outcomes of the different kernels for varying bandwidths in your report. Last, run `test_kde_1d_fewsamples`, which has data gaps. Again, compare all results in your report and explain reasons for performance differences (between kernels, bandwidths, and compared to complete data samples of `test_kde_1d`) based on the theory.

Exercise 3. 5 P.

Write a function `ii_meanshift` that performs the mean shift algorithm. The function takes 3 arguments:

1. input data: sample points in a N-by-2 matrix (number of rows is the number of samples, dimensionality of the input data will always be 2 for this exercise)
2. the kernel bandwidth h
3. the stopping threshold ϑ

It should return two values.

1. cluster indexes: a column vector with N rows, specifying the cluster index for each sample
2. cluster modes: a M-by-2 matrix, returning the cluster modes (the points with the highest density) for each cluster (where M is the number of clusters)

For this task, use the Epanechnikov kernel. Luckily all terms before the sum cancel out in the mean shift formula, leading to

$$\mathbf{q}_{t+1} = \frac{\sum_{i=1}^{N} \mathbf{x}_i \max\left(0, 1 - \frac{\|\mathbf{q}_t - \mathbf{x}_i\|^2}{h^2}\right)}{\sum_{i=1}^{N} \max\left(0, 1 - \frac{\|\mathbf{q}_t - \mathbf{x}_i\|^2}{h^2}\right)} \tag{5}$$

Start the mean shift procedure at each point and iterate until $\|\mathbf{q}_t - \mathbf{q}_{t-1}\| < \vartheta$ where ϑ is the threshold passed to the function. Alternatively, stop the iteration when $\frac{s_t}{s_{t-1}} < 10^{-6}$ where $s_t = \|\mathbf{q}_t - \mathbf{q}_{t-1}\|$ (which works slightly better and removes the need for the threshold θ). When the iteration stopped, decide if a cluster mode already exists that is closer than $\frac{h}{5}$. If yes, assign the point that you started at to this cluster. Otherwise, create a new cluster and assign the point to the new cluster. It can be helpful to visualize the steps you are taking during the mean shift procedure to spot errors.

Report: Evaluate your code with test scripts `test_meanshift1` and `test_meanshift2`. The first test script `test_meanshift1` uses exactly the same data as `test_kmeans1` for k-means. Compare results of k-means and mean shift and explain performance differences based on the theory behind both algorithms. Finally, run `test_meanshift2`, which samples data points randomly like `test_kmeans2` for k-means, and again compare both k-means's and mean shift's results in your report. Also try `test_meanshift2` multiple times for fixed mean shift parameter settings and report your findings.