Machine Learning Homework #6

Clustering – Gerald Baulig – 0780827

This is the report of Machine Learning Homework #6, documentation of the code and discussion of the results.

Remarks: I recommend to use any other video-player then the Microsoft-APPs. There are no video-clips of Spectral Clustering because there is nothing to animate beside the KMeans separation of the eigenvectors, which terminates in almost 2 steps. Instead a GIF is provided that demonstrates the effect of gamma of the RBF-kernel.

1 Questionnaire

The following sections answer the questionnaires of the tasksheet.

1.1 You need to make videos or GIF images to show the clustering procedure (visualize the cluster assignments of data points in each iteration, colorize each cluster with different colors) of your k-means, kernel k-means, spectral clustering and DBSCAN program.

Please check the folder named ‘media’ of this project. The annotation of these clips describes the content as follows:

<cluster method>_k<number of clusters>_<optional parameters>_<the dataset>

Here some examples:

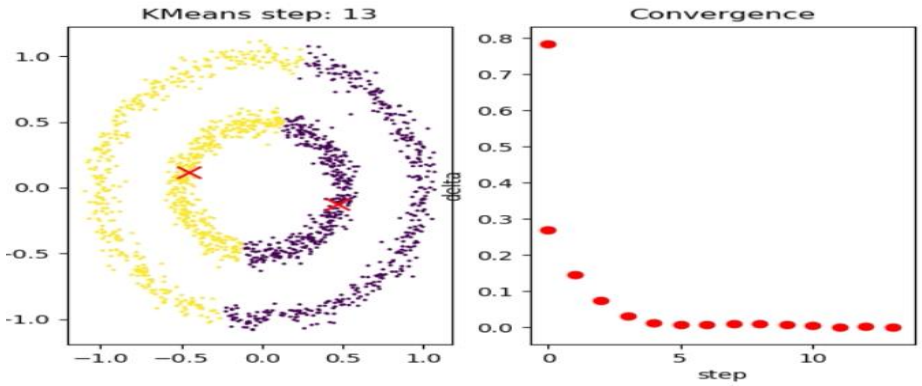


Fig 1: **kmeans_k2_select_circle.mp4** – Shows a KMeans clusters the circle.txt dataset into 2 clusters initialized by random select mode. Mind the fast convergence. Nevertheless, simple KMeans will not be able to cluster connected shapes like in ‘circle.txt’ or ‘moon.txt’.

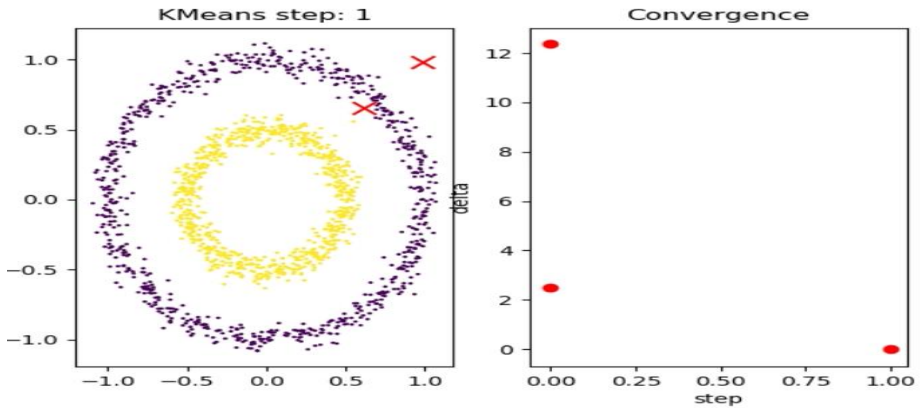


Fig 2: **kmeans_k2_select_RBF_circle.mp4** – Shows a Kernel KMeans clusters the circle.txt into 2 clusters using an RBF-kernel initialized by random select mode. While simple KMeans is not able to cluster connected shapes properly, Kernel-KMeans does since all datapoints are translated to a high-dimensional feature-vector of a gram-matrix.

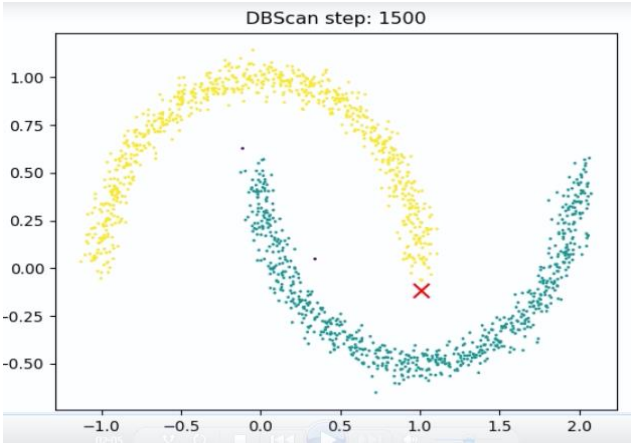


Fig 3: **dbscan_10p_r10e-2_moon.mp4** – Shows a DBScan with minimal points of 10 in a radius of 10e-2 (0.1) applied on the moon.txt dataset. The nature of this algorithm is to follow along connected shapes. Hence DBScan solve ‘circle.txt’ and ‘moon.txt’ pretty well. However, in the naïve path-following-approach DBScan needs to check almost every point, such that no path gets lost.

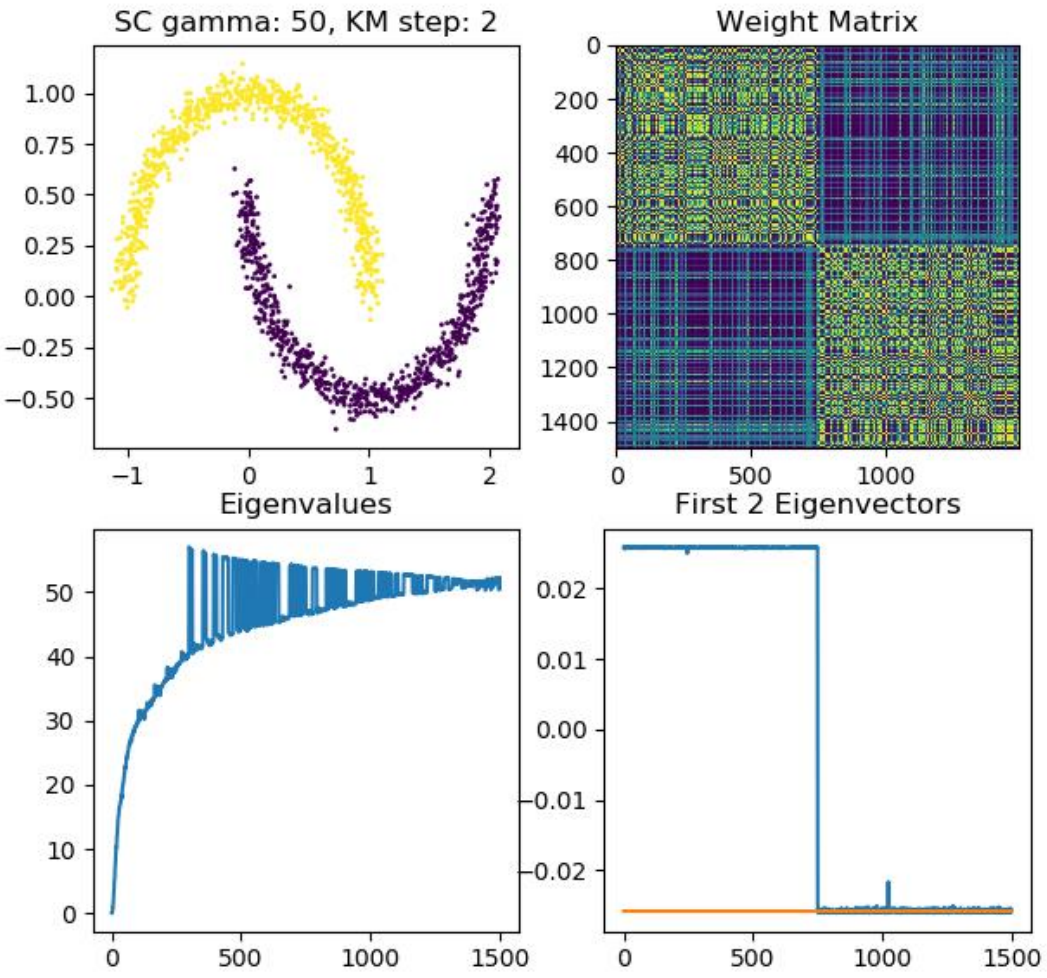
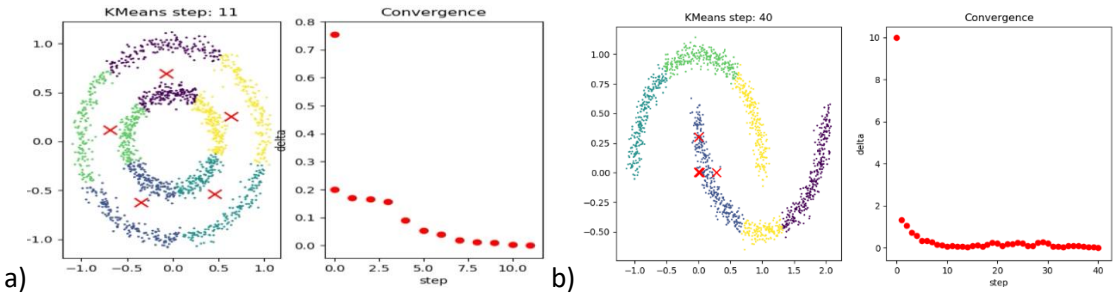


Fig 4: **spectral_k2_g50_moon.png** – Spectral Clustering clusters the 'moon.txt' into 2 clusters with a gamma of 50. Upper-right the Weight- or Similarity-Matrix is visualized. Lower-left shows all Eigenvalues of the Laplacian-Matrix. Since this example works with fully-connected graphs only the eigenvalue_0 is close to 0 (not exactly 0 because of floating point errors!). This indicates a fully-connected component. To indicate 2 clusters the eigenvector_1 is relevant, as illustrated lower-right (blue = eigenvector_1, orange= eigenvector_0)

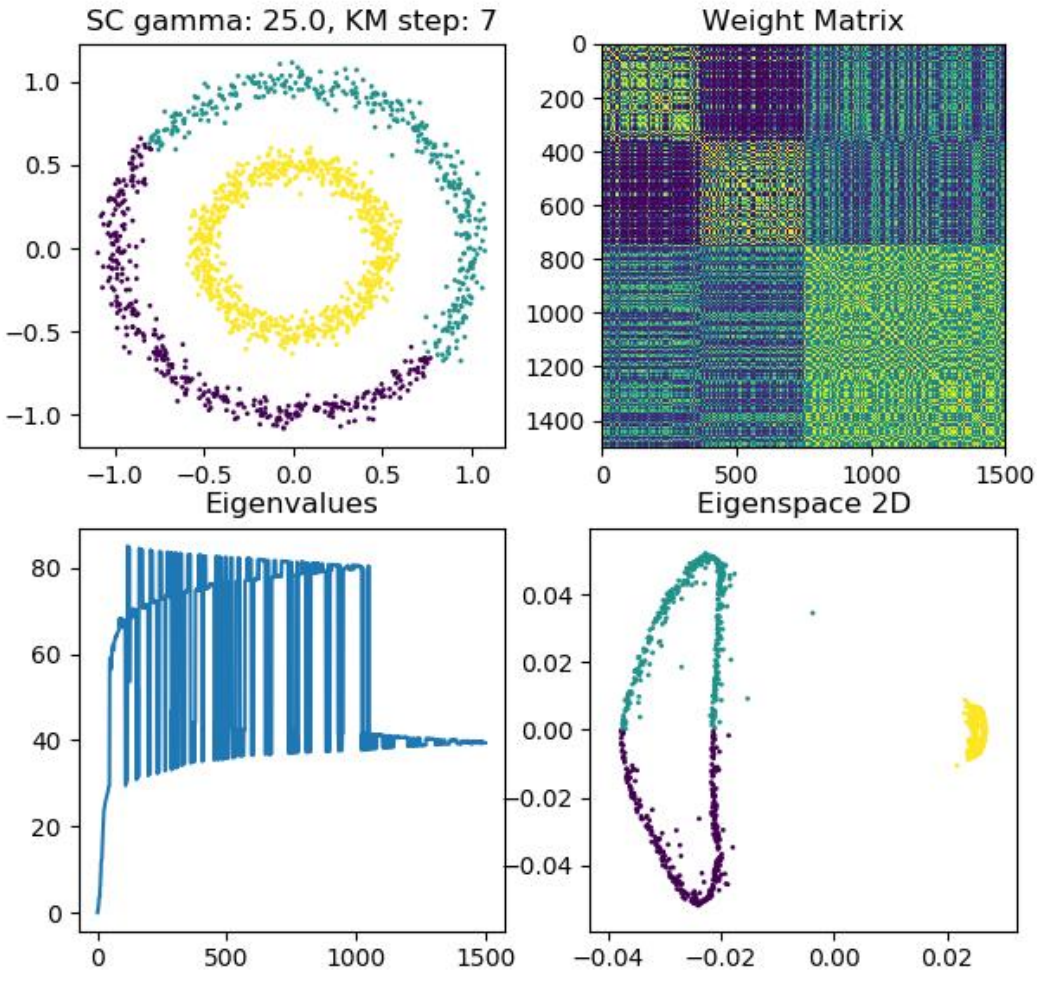
Feel free to explore further medias!

1.2 In addition to cluster data into 2 clusters, try more clusters (e.g. 3 or 4) and show your results.

KMeans, Kernel-KMeans, and Spectral-Clustering require an initial number of clusters they have to build, while DBScan explore the cluster-number by itself. Hence there is no reason to prime a cluster-number in DBScan. The following 3 clips show KMeans, Kernel-KMeans and Spectral-Clustering clustering the circle dataset into 5 clusters:



a) b)



c)

Fig 5: a) Simple KMeans clusters ‘cirlce.txt’ into 5 cluster initialized via kmeans++, b) Kernel-KMeans clusters ‘moon.txt’ into 5 clusters by using the RBF kernel function and initialized via kmenas++, c) Spectral Clustering clusters the ‘circle.txt’ into 3 clusters by using RBF as similarity function with gamma 50.

In simple KMeans it can be observed that the cluster-centers swarm out and distribute uniform. Kernel-KMeans has a similar behavior. Mind that the markers for the cluster-centers have no relevance, since the visualization of 1500-dimansions is barely possible. For Spectral Clustering the 2D-Eigenspace reveals how the clustering of two circles works.

Please check the corresponding clips.

1.3 For the initialization of k-means clustering used in k-means, kernel k-means and spectral clustering, try different ways and show corresponding results, e.g. k-means++.

Spectral-Clustering is originally just transforming the given dataset into a high-dimensional feature-space build by the eigenvectors of a Laplacian Matrix. After this step a separation function is still required to cluster the high-dimensional feature-vectors. This could be done via SVM or KMeans. In this project KMeans is employed. Hence, the initialization methods of KMeans can be applied on KMeans itself, Kernel-KMeans and Spectral-Clustering.

In this project we implemented 5 initialization modes:

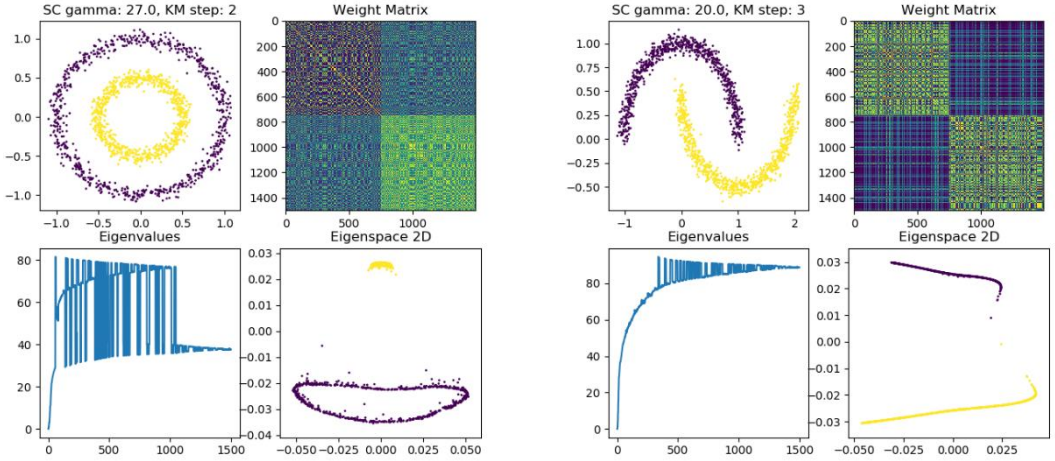
1. **Mean**: All cluster centers are initialized close to the mean of the whole dataset. However, a small portion of the dataset's variance is applied as a small error. Otherwise, if all cluster centers would be initialized on the perfect mean of the dataset, the first cluster center would claim all points while it keeps its position and the algorithm converges because the update delta is 0.0.
2. **Uniform**: All cluster centers are initialized via uniform random generator, but bounded to the range of the dataset.
3. **Normal**: All cluster centers are initialized via normal random generator, but bounded to the range of the dataset by taking the mean and the variance of the dataset.
4. **Select**: Selects one random point from the dataset for each cluster center to initialize.
5. **KMeans++**: A complex approach introduced by [XXX]. Selects a random point from the dataset and arranges the probabilities for the next random point depending on the distance. Points far from the currently selected point have a higher probability of being selected next, until K points are selected. This may lead to a well distributed set of cluster centers.

Please check the following clips:

kmeans_k5_mean_circle.mp4: Simple KMeans clustering the circle.txt dataset into 5 clusters initialized via Mean mode. This clip demonstrates, even if all the cluster centers are initialized at the same point, they are still able to swarm out, as long they are not initialized at the perfect mean or beyond the dataset range.

kmeans_k5_pp_circle.mp4: Simple KMeans clustering the circle.txt dataset into 5 clusters initialized via the KMeans++ algorithm. Against the statements of the D. Arthur et al. [1] KMeans++ is not outperforming the other initialization methods.

1.4 For spectral clustering, you can see if data points within the same cluster do have the same coordinates in the eigenspace of graph Laplacian. You should plot the result and discuss it in the report.



In these two plots of Spectral Clustering the 2D Eigenspace is illustrated in the 4th subplot (lower-right). These plots are created by taking the two first non-zero Eigenvectors and scattered on a 2D plane. It clearly points out how the points got rearranged depending on the similarity weights of the fully-connected graph. Points that are originally close to each other with a high degree stay close to each other. Points that are originally far from each other get pushed away from each other into the new generated dimensions. The higher we choose gamma, the sharper the points get separated.

2 Code Explanation

This project includes 3 executable demo scripts and 3 helper modules. All the clustering algorithms are implemented in 'clustering.py'. The 'kernel.py' provides kernel and similarity functions. The 'utils.py' provides helper functions for user interactions and visualization.

2.1 KMeans & Kernel-KMeans

For KMeans or Kernel-KMeans the 'demo_kmeans.py' is to execute. Use the command '-h' to see an explanation of the optional parameters:

```
>python demo_kmeans.py -h
usage: demo_kmeans.py [-h] [--data FILEPATH] [--centers INT] [--epsilon INT]
                      [--mode {mean,select,uniform,normal,kmeans++}]
                      [--kernel {eucledian,linear,RBF,linearRBF,none}]
                      [--params ARGS] [--video FILEPATH]

Demo for clustering via (Kernel-)KMeans

optional arguments:
  -h, --help            show this help message and exit
  --data FILEPATH, -X FILEPATH
                        The filename of a csv with datapoints.
  --centers INT, -k INT
                        The number of cluster centers.
  --epsilon INT, -e INT
                        The convergence threshold.
  --mode {mean,select,uniform,normal,kmeans++}, -m {mean,select,uniform,normal,kmeans++}
                        Choose an initialization mode.
  --kernel {eucledian,linear,RBF,linearRBF,none}, -K
{eucledian,linear,RBF,linearRBF,none}
```

```
                        Choose a kernel for kernel KMeans.
  --params ARGS, -p ARGS
                        Parameters for the kernel, if required. E.g:
                        'gamma=1.0, sigma=0.5'
  --video FILEPATH, -V FILEPATH
                        A filename for video record.
```

Epsilon is the threshold that leads the KMeans algorithm to converge. However, epsilon=0 is recommended, since it leads to a more accurate result and KMeans converges quickly anyway.

The core functionality of KMeans is implemented in 'clustering.py'. A function that returns a generator that can be used as an iterator:

```
def kmeans(X, means, epsilon=0.0, max_it=1000, is_kernel=False):
    '''…'''
    N = X.shape[0]      #N: Size of the dataset
    K = means.shape[0]  #K: Number of clusters
    W = np.zeros((N,K)) #W: Distance (Weight) matrix
    C = np.zeros((N,K)) #C: Association matrix
    delta = None

    if is_kernel:
        for k, m in enumerate(means):
            W[:,k] = square_mag(X, m) #calc square distances
        Y = np.argmin(W, axis=1) #find closest
        C[:,Y] = 1 #set association flags

    for step in range(max_it):
        if is_kernel:
            W = kernel_trick(X,C)
        else:
            for k, m in enumerate(means):
                W[:,k] = square_mag(X, m) #calc square distances

        Y = np.argmin(W, axis=1) #find closest

        tmp = means.copy()
        for k in range(K):
            C[:,k] = Y==k #set association flags
            if any(C[:,k]): #only update if associations are given
                means[k,:] = np.mean(X[Y==k,:], axis=0) #calc the means

        delta = np.sum((tmp - means)**2) #calc the delta of change
        yield Y, means, delta, step #yield for mean update

        if delta <= epsilon:
            break #converge if change event is lower-equal than epsilon
    pass
```

For simple KMeans the distance is computed via the squared magnitude between mean-point and datapoint, also known as the Euclidean Distance of two vectors. Let i= the row index, d= the dimension, u= the datapoints and v= the current mean:

$$y_i = \sum_d (u_{id} - v_d)^2$$

```
def square_mag(u, v):
    '''…'''
    return np.sum((u-v)**2, axis=1)
```

In the case of Kernel-KMeans, the distances are computed via Gram-Matrix of a kernel function and with the help of the kernel-trick:

$$\mathrm{k}(x_j, x_j) - \frac{2}{|C_k|}\sum_n \alpha_{kn}\mathrm{k}(x_j, x_n) + \frac{1}{|C_k|^2}\sum_p\sum_q \alpha_{kp}\alpha_{kq}\mathrm{k}(x_p, x_q)$$

```
def kernel_trick(gram, C):
    '''…'''
    N = gram.shape[0]
    K = C.shape[1]
    A = np.sum(C,axis=0)
    ones = np.ones((N, K))

    W = np.dot(gram * np.eye(N), ones)
    W -= 2*(np.dot(gram, C) / A)
    W += np.dot(np.dot(np.dot(C.T, gram), C) * np.eye(K), ones)
    W /= (A**2)
    return W
```

2.2 The Kernels

In this project 5 kernels are implemented: 'euclidean', 'linear', 'RBF' and 'linearRBF'.

The **Euclidean** kernel computes the distances between each point of 2 matrices:

$$k(u, v) = |u - v^T|^2$$

```
def euclidean(u, v, params={}):
    '''…'''
    if len(u.shape) == 1:
        u = u[:,None]

    if len(v.shape) == 1:
        v = v[:,None]

    return np.sum(u**2, axis=1) + np.sum(v**2, axis=1)[:,None] - 2*np.dot(u,v.T)
```

The **Linear** kernel can be described as:

$$k(u, v) = uv^T$$

```
np.dot(u, v.T)
```

So the **RBF** kernel:

$$k(u, v) = \exp(\gamma\ |u - v|^2)$$

```
np.exp(-g * euclidean(u,v))
```

And the linearRBF kernel:

$$k(u, v) = uv^T + \exp(\gamma\ |u - v|^2)$$

```
linear(u, v) + RBF(u, v, params)
```

In this report only the RBF kernel is demonstrated. Feel free to experiment with the others.

2.3 Initialization of KMeans

In this project 5 initialization modes are implemented: ‘mean’, ‘uniform’, ‘normal’, ‘select’ and ‘kmeans++’. Please check following code and the pros and cons in the documentation string:

```
def init_kmeans(X, K, mode='mean'):
    ''' init_kmeans(X, K, mode='free') -> means
    Initialize K cluster means on dataset X.

    Args:
        X: The dataset.
        K: The number of cluster means.
        mode: The mode how to initialize the cluster means.
            mean = All means start (almost) at the mean of the dataset.
                Pro: The result is deterministic.
                Con: Needs more iterations.
            uniform = Uniform random distributed.
                Pro: May work well on uniform distributed datasets.
                Con: Ks may get lost in huge data gaps.
            normal = Normal random distributed.
                Pro: May work well on normal distributed datasets.
                Con: Ks may get lost in huge data gaps.
            select = Selects random points of the dataset.
                Pro: Makes sure that each K has at least one point.
                Con: Not deterministic like all the other random approaches.
            kmeans++ = Selects a random point and rearranges the
                probabilities of selecting the next point according
                to the distance.
                Pro: May have an appropriate distributed of Ks.
                Con: Great effort for a negligible improvement.
    Returns:
        means: The cluster means.
    '''
    N = X.shape[0]
    d = X.shape[1]

    if mode=='mean':
        #One can not take the absolute mean.
        #The first K would claim all points and the algo stuck.
        #Get the means and add a small portion of the variance.
        #This ensures a small error and the Ks will swarm out.
        #Pro: The result is determenistic.
        #Con: Needs more iterations.
        means = np.tile(np.mean(X, axis=0), (K,1)) + np.var(X, axis=0) * 1e-10
    elif mode=='select':
        means = X[np.random.choice(range(N), K),:]
    elif mode=='uniform':
        means = np.random.rand(K,d) * np.var(X, axis=0) + np.mean(X, axis=0)
    elif mode=='normal':
        means = np.random.randn(K,d) * np.var(X, axis=0) + np.mean(X, axis=0)
    elif mode=='kmeans++':
        #guided by https://stackoverflow.com/questions/5466323/how-exactly-does-k-means-
work
        means = np.zeros((K,d))
        means[0,:] = X[np.random.choice(range(N), 1),:] #pick one
        for k in range(1,K):
            W = square_mag(X, means[k-1]) #get the distances
            p = np.cumsum(W/np.sum(W)) #spread probabillities from 0 to 1
            i = p.searchsorted(np.random.rand(), 'right') #pick next center to random
distance
            means[k,:] = X[i,:]
    else:
        raise ValueError("Unknown mode!")
    return means
```

The great advantage of ‘mean’ is its deterministic behavior. With the same parameter-set it will always conclude to the same result. The random approaches ‘uniform’ and ‘normal’ are having both the disadvantage that cluster-centers may get lost in huge date gaps, such that no datapoint gets assigned to them. The ‘select’ ensures that each center will have at least one datapoint, but it may happen that more cluster-centers will drop into one isolated data island, while other centers claim several islands to its own. ‘kmeans++’ tries to avoid this issue by rearranging the probabilities according to the distance from the latest initialized cluster-center. General speaking: it tries to keep the cluster-centers far from each other.

However, the implementation is naïve, followed by [1]. It might be more effective to all initialized cluster-centers in count, to place the next one as most far from them, for a well distributed start configuration.

2.4 DBScan

The DBScan algorithm is a simple naïve approach, were an agent scans paths along the dataset and unites points to a determinate density in an expected radius. Please check the help message of ‘demo_dbscan.py -h’ for the explanation of optional parameters:

```
>python demo_dbscan.py -h
usage: demo_dbscan.py [-h] [--data DATA] [--min_points MIN_POINTS]
                      [--radius [RADIUS [RADIUS ...]]] [--video VIDEO]

Demo for clustering via DBScan

optional arguments:
  -h, --help            show this help message and exit
  --data DATA, -X DATA  The filename of a csv with datapoints.
  --min_points MIN_POINTS, -p MIN_POINTS
                        The number minimal points for cluster growing.
  --radius [RADIUS [RADIUS ...]], -r [RADIUS [RADIUS ...]]
                        The radius for cluster growing. Can be multy-
                        dimensional.
  --video VIDEO, -V VIDEO
                        A filename for video record.
```

The core function is implemented in ‘clustering.py’. This function returns a generator to iterate through the cluster steps. The path-following-version of DBScan has to chack almost every datapoint, such that no path may get lost:

```
def dbscan(X, points, radius):
    ''' … '''
    N = X.shape[0]
    Y = np.zeros(N)
    radius = radius**2 #lets take radius in power of 2 since we use square_mag
    C = 1 #the current cluster label
    step=0

    def pick_next():
        ''' pick_next() -> yields i
        A generator to pick next unlabeled point.

        Yields:
            i: The next index of unlabeled datapoint.
        '''
        while True:
            i = np.nonzero(Y==0)[0]
            if i.size == 0:
                break
            else:
```

```
                yield i[0]
        pass

    def scan(i):
        ''' scan(i) -> yields n
        The core function of dbscan.
        Check the given point under index i to be:
        core-point, border-point, or noice.

        Args:
            i: The index of the point to be scanned.
        Returns:
            n: Indices of processable neighbors.
        '''
        n = square_mag(X, X[i]) <= radius
        #Do not count the current core point.
        if sum(n) > points: #density sufficient
            #check for unprocessed points and give noise a second chance.
            n = np.logical_and(n, np.logical_or(Y==0, Y==-1))
            n[i] = False
            Y[n] = C
            return np.where(n)[0]
        else: #density not sufficient
            n = np.logical_and(n, Y==0)
            Y[n] = -1 #label as noise
            return np.ndarray(0)
        pass

    for i in pick_next():
        F = [i] #Here we start a new cluster and a new FIFO
        for i in F: #Pop from FIFO
            F.extend(scan(i).tolist()) #Push new datapoints to be scanned
            step += 1
            yield Y, X[i], step #Yields labels, current position and step counter
        C += 1 #Cluster complete, proceed to next label
    Pass
```

It points out that the search radius is not well utilized. Points in the radius with sufficient density get labels. Nevertheless, for the path finding they still needs to be visited, but while doing this the agent uses just a sickle shape of the given radius. Hence, the agent approaches in small steps.

Instead of a pathfinding-approach I would suggest a ‘join-approach’:

1. Choose a random point.
2. Scan the area like before (DBScan).
    a. If none of these points is already labels, assign a new label to them.
    b. If some points have already a label, but all of them the same one, label all the others too.
    c. If several labels are in the scan-radius, then relabel all the points that ever got these labels with the lowest one (the join-approach).
3. If any point is still unlabeled, return to 1. Otherwise quit.

This approach utilizes the search-radius to a higher ratio and it is much easier to translate this algorithm to high parallel processing.

3 Spectral Clustering

The Spectral Clustering is a complex algorithm that employs the eigenvectors of a Laplacian-Matrix generated by the Similarity-Matrix of a fully-connected graph. Hence, the graph needs to be built the get the Laplacian-Matrix and finally the eigenvectors. The eigenvectors can be separated via i.e.: SVM

or KMeans. KMeans is preferred and implemented in this project. Please check the help message of 'demo_spectral.py -h' to get explanations of optional parameters:

```
> python demo_spectral.py -h
usage: demo_spectral.py [-h] [--data DATA] [--centers CENTERS] [--gamma GAMMA]
                        [--epsilon EPSILON] [--laplacian_mode LAPLACIAN_MODE]
                        [--kmeans_mode {mean,select,uniform,normal,kmeans++}]

Demo for clustering via Spectral Clustering

optional arguments:
  -h, --help            show this help message and exit
  --data DATA, -X DATA  The filename of a csv with datapoints.
  --centers CENTERS, -k CENTERS
                        The number of cluster centers.
  --gamma GAMMA, -g GAMMA
                        The energy factor of similarity.
  --epsilon EPSILON, -e EPSILON
                        The edge weight threshold for partially connected
                        graphs.
  --laplacian_mode LAPLACIAN_MODE, -L LAPLACIAN_MODE
                        The mode of how to compute Laplacian Matrix.
  --kmeans_mode {mean,select,uniform,normal,kmeans++}, -K
{mean,select,uniform,normal,kmeans++}
                        The mode of how to initialize KMeans.
```

In this project 3 Laplacian computation modes are implemented: 'default', 'shi' and 'jordan'. Default is the unnormalized Laplacian-Matrix:

$$L = D - W$$

The 'shi' mode is a normalization ala Shi et al. [2]:

$$L = D^{-0.5} L D^{-0.5}$$

The 'jordan' mode is a normalization ala Ng and Jordan [3]:

$$L = D^{-1} L$$

But mainly the unnormalized version is sufficient for the datasets of this project.

If epsilon is greater than zero, the Spectral Clustering switch to a discrete mode. In this mode the Degree-Matrix count the K-Nearest-Neighbors (knn), instead of sum up all weights in a continuous mode.

The Spectral Clustering algorithm is implemented as a class. The core functions are a set-method to update the matrices as required, and the cluster function that returns a generator like before:

```
def set(self, X=None, gamma=None, epsilon=None, mode=None):
        ''' set(X=None, gamma=None, epsilon=None, mode=None)
        Use this function to change one or several properties.
        Recomputes the depending properties as required.

        Args:
            X: Set the dataset - updates the whole spectral information
            gamma: Set gamma - updates W, D, L, eigval and eigvec
            epsilon: Set epsilon - updates D, L, eigval and eigvec
            mode: Set the mode - updates L, eigval and eigvec
        '''
        recalc = False
        if isinstance(X, np.ndarray):
            self.__X = X
```

```
            self.__N = X.shape[0]
            self.__d = X.shape[1]
            recalc = True

        if gamma != None:
            self.__gamma = gamma
            recalc = True

        if recalc:
            self.__W = kernel.RBF(self.__X, self.__X, {'gamma':gamma})

        if epsilon != None:
            self.__epsilon = epsilon
            recalc = True

        if recalc:
            if self.__epsilon:
                self.__D = np.sum(self.__W > self.__epsilon, axis=1)*np.eye(self.__N)
            else:
                self.__D = np.sum(self.__W, axis=1)*np.eye(self.__N)

        if mode != None:
            self.__mode = mode
            recalc = True

        if recalc:
            L = self.__D - self.__W
            if mode == 'shi':
                D = np.sqrt(self.__D)
                D = np.linalg.inv(self.__D)
                self.__L = D*L*D
            elif mode == 'jordan':
                D = np.linalg.inv(self.__D)
                self.__L = D*L
            elif mode == 'default':
                self.__L = L
            else:
                raise ValueError("Unknown mode!")

            self.__eigval, self.__eigvec = np.linalg.eig(L)
        pass
```

After setting the parameters, the cluster function can be called to iterate though the KMeans separation:

```
def cluster(self, K, mode='select'):
        ''' cluster(K, mode='select') -> kmeans generator
        Initialize and returns a KMeans generator.
        (See kmeans)

        Args:
            K: Number of cluster centers
            mode: The initialization mode (See init_kmeans)
        Returns:
            Generator of KMeans, yields:
                Y: The labels
                means: The cluster centers
                delta: The update delta
                step: The update step counter
        '''
        X = self.__eigvec[:,0:K]
        means = init_kmeans(X, K, mode)
        return kmeans(X, means)
```

References

[1] Arthur, D.; Vassilvitskii, S. (2007)."*k*-means++: the advantages of careful seeding". *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics Philadelphia, PA, USA. pp. 1027–1035

[2] J. Shi, J. Malik: ”*Normalized cuts and image segmentation.*” In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 22(8), (2000), S. 888–905. doi:10.1109/34.868688

[3] A. Y. Ng, M. I. Jordan, Y. Weiss: ”*On spectral clustering: Analysis and an algorithm.*” In: *Advances in Neural Information Processing Systems.* 2, (2002), S. 849–856.