sheet02_final

November 7, 2022

1 Sheet 2 - Bhavesh Rajpoot(wk282), Simran Joharle(vz282), Sachin Gupta(vl282)

[ ]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

plt.rc('font', family='monospace', size=12)
plt.rc('mathtext', fontset='stix')



from scipy.stats import gaussian_kde
```

##

1. Kernel Density Estimation

1.0.1 (a) Implement a Quartic (biweight) function:

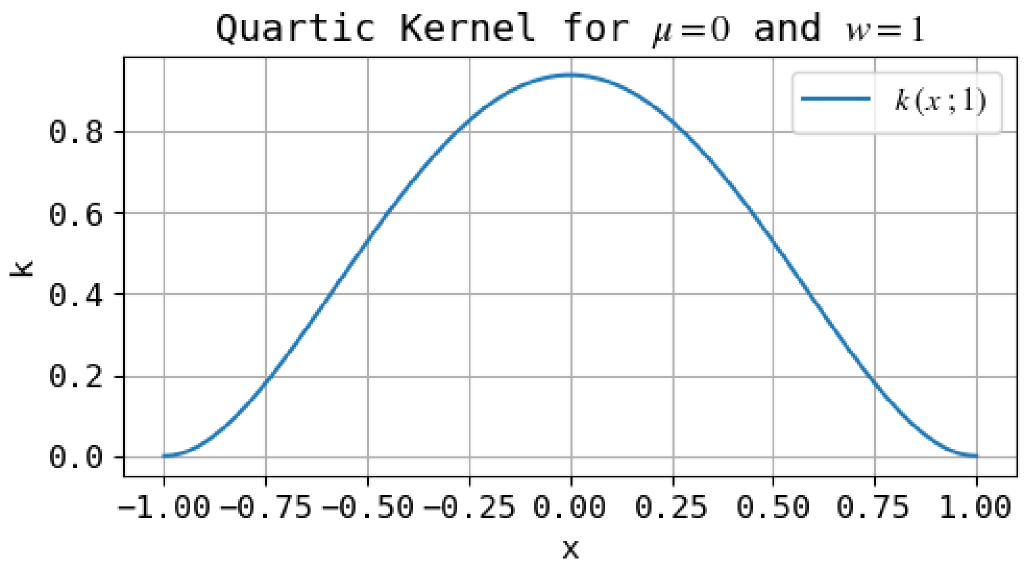
$$k(x-\mu; w) = \frac{15}{16w}\left(1-\left(\frac{x-\mu}{w}\right)^2\right)^2 \qquad \text{with support in } [-w, w]$$

substituting $u = \frac{x-\mu}{w}$ for generalising the equation:,

$$k(u) = \frac{15}{16w}\left(1-(u)^2\right)^2, \qquad for\ |u| \leq w$$

[ ]:
```
 # defining kernel class
class kernels:
    '''
    A class to represent a set of kernels for KDE
    ...
    Attributes
    ----------
    x : float or nd.array
    mean : float or nd.array
    bandwidth : float
    Methods
    -------
    biweight(self):
        returns biweight kernel

    gaussain(self):
        returns gaussian kernel

    epa(self):
        returns epanechnikov kernel
    '''
    def __init__(self,x,mean,bandwidth): #initialising setp
        self.x = x
        self.mu = mean
        self.w = bandwidth
        self.u = (x-mean)/bandwidth

    # defining biweight kernel
    def biweight(self):
        """biweight kernel at mean mu, with bandwidth w evaluated at x"""
        #TODO: implement the quartic (biweight) kernel
        k=[]
        for i in range(0,len(self.x)): #if else condition for implementing the␣
↪support in [-w,w]
            if(abs((self.x[i]-self.mu)/self.w)<=self.w):
                k.append((15/(16))*((1-((self.x[i]-self.mu)/self.w)**2)**2))
            else:
                k.append(0)

        return np.array(k)

    # defining  gaussian kernel
    def gaussian(self):
        """gaussian kernel at mean mu, with bandwidth w evaluated at x"""
        return (1/((2*np.pi)**0.5))*np.exp(-0.5 * self.u**2)

    # defining epanechnikov kernel
    def epa(self):
        """epanechnikov kernel at mean mu, with bandwidth w evaluated at x"""
        k=[]
        for i in range(0,len(self.x)):
            if(abs((self.x[i]-self.mu)/self.w)<=1):
                k.append((3/4)*((1-((self.x[i]-self.mu)/self.w)**2)))
            else:
                k.append(0)
        return np.array(k)
```

```
[ ]: # TODO plot the kernel

     x = np.linspace(-1,1,100)
     k = kernels(x,0,1).biweight() #initialising the kernel for mu=0, and w=1 over␣
      ↪the range x=[-1,1]

     #plotting the result
     fig = plt.figure(figsize=(6,6))
     ax = fig.gca()
     plt.plot(x,k,label=r'$k\;(x\;;1)$')
     plt.xlabel('x')
     plt.ylabel('k')
     plt.title(r'Quartic Kernel for $\mu=0$ and $w=1$')
     plt.legend()
     ax.set_aspect(1)
     plt.grid()
```



1.0.2 (b) Take the first $N = 50$ data points from samples.npy, compute and plot the KDE over the range $[-10, 20]$ for a set of different bandwidths (e.g. $w \in 0.1, 0.5, 1, 3, 5$). Discuss the results and the influence of the bandwidth. Which bandwidth is optimal in your opinion? Explore what happens as you increase the number of samples N.

```
[ ]: # load the data
     data = np.load("data/samples.npy")
     data50 = data[:50]
     print(f'{data.shape=}, {data50.shape=}')
```

```
data.shape=(10000,), data50.shape=(50,)
```

In order to get the Kernel Density Estimate, we need to imply the following algorithm over the given dataset.

KDE Algorithm:

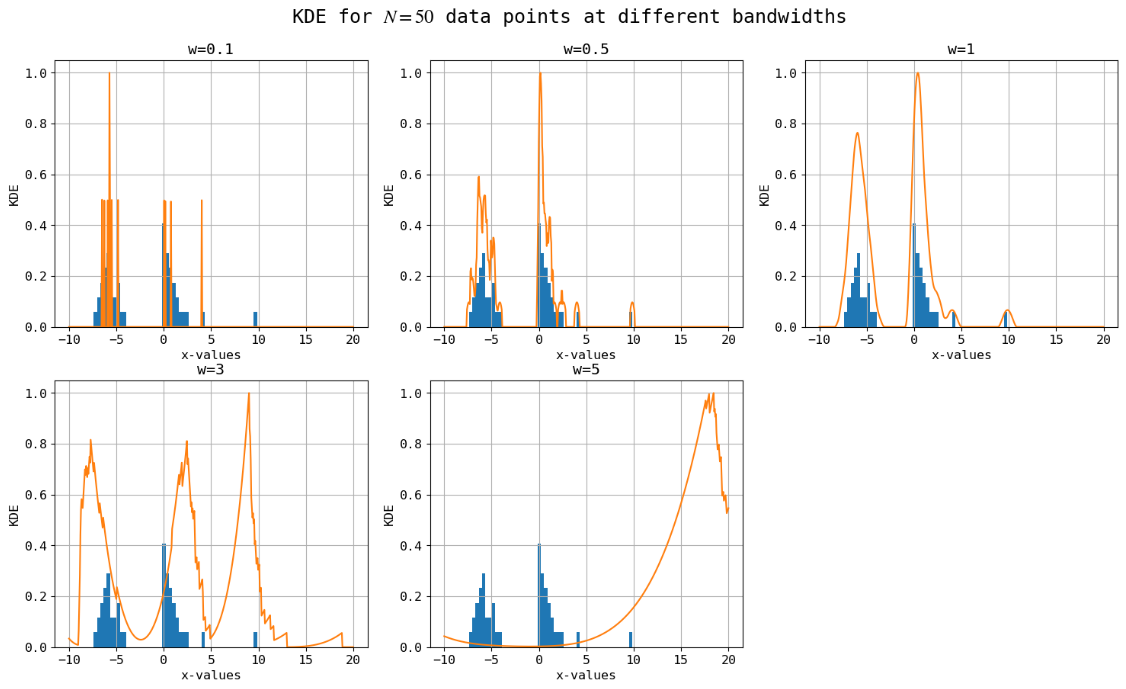
$$f(x) = \frac{1}{nw} \sum_{i=0}^{n} K\left(\frac{x - X_i}{w}\right)$$

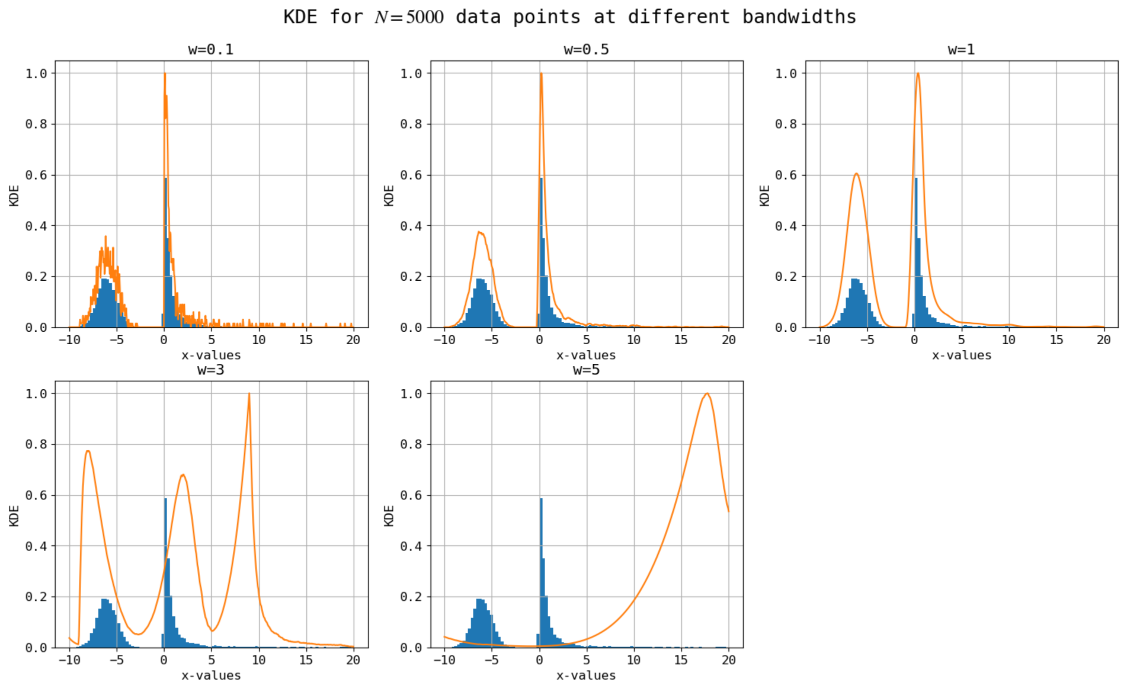
where, - K = Kernel function - n = no. of observations - X_i = i^{th} observation of the random variable - w = bandwidth

```
[ ]: def kde(x, obs, w=1):
         # TODO: implement the KDE with the biweight kernel
         n = len(obs)
         density = (sum(kernels(x, xi, w).biweight() for xi in obs))/(n*1) #summing␣
      ↪the kernels over each observation to get the complete KDE

         return density

     def kde_plotter(data,x,w,ncols,nrows=None,bins=50,figsize=(18,10)):
         '''
         Plots KDE over given dataset
         '''
         plt.figure(figsize=figsize)
         plt.subplots_adjust(hspace=0.2)
         plt.suptitle(f"KDE for $N={len(data)}$ data points at different␣
      ↪bandwidths", fontsize=18, y=0.95)
         plt.tight_layout()

         #setting no. of rows and columns for subplot
         ncols = 3
         if nrows == None:
             nrows = len(w) // ncols + (len(w) % ncols > 0) # calculating number of␣
      ↪rows
```

```
    for n,wi in enumerate(w):
        #adding subplot iteratively
        ax = plt.subplot(nrows, ncols, n + 1)

        #plotting the data histogram and KDE
        ax.hist(data,bins=bins,density=True);
        fx = kde(x, data, wi)
        ax.plot(x,fx/np.max(fx))

        # chart formatting
        ax.grid()
        ax.set_title(f'w={wi}')
        ax.set_xlabel("x-values")
        ax.set_ylabel("KDE")
```

[ ]:
```
# TODO: compute and plot the kde on the first 50 data points

x = np.linspace(-10,20,500) # x-array
w = [0.1,0.5,1,3,5] #list of weights to implement

kde_plotter(data50,x,w,3,bins=50,figsize=(18,10))

# TODO: explore what happens when you increase the number of points
data5000 = data[:5000]
kde_plotter(data5000,x,w,3,bins=100,figsize=(18,10))
```





1.1 Discussion

1.2 On Bandwidth:

1. Bandwidth definately changes a lot how the KDE looks like.
2. Smaller bandwidths leads to undersmoothing and larger bandwidths leads to oversmoothing.
3. Moreover, apart from oversmoothing on larger bandwidths, we can observe extra peaks in the KDE mainly in $w = 3$ and $w = 5$ plots.
4. To inspect the reason behind that, we plotted the base Kernel for varying bandwidths and found out that the Quartic function has side peaks for $w > 1$ (see the plots below). These side peaks are the reason for bad KDE in our plots.
5. Hence, from this we can also confirm that $u = \frac{x-\mu}{w}$ should always follow $|u| \leq 1$. This condition is aso true for epanechnikov kernel.

1.3 On optimal bandwidth:

1. From the plots, we arrived at the conclusion that $w = 1$ is the most optimal bbandwidth for this dataset as it has clearly spereated smoothed peaks.
2. Smaller bandwidths produce noisy KDEs and hence not an optimal choice.
3. More precise bandwidth for a particular dataset can be selected using advanced methods like Cross-Validation, Sheather and Jones method, etc.

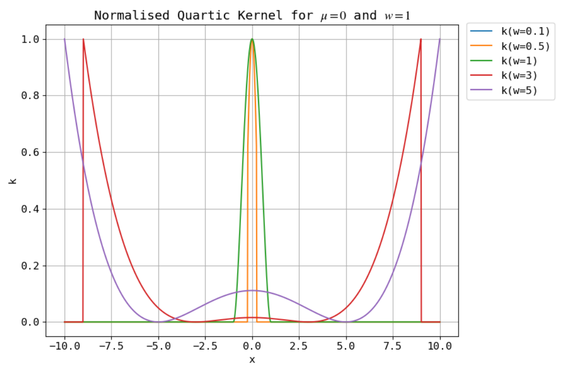
1.4 On increased no. of samples N:

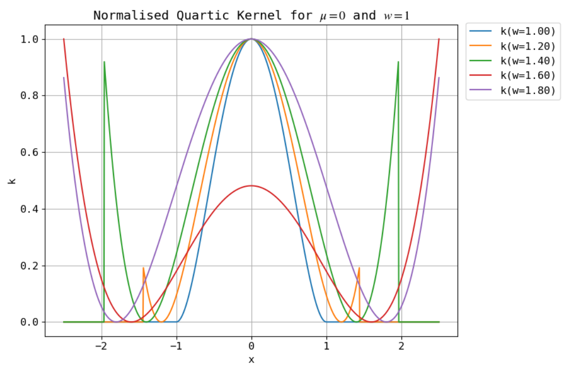
1. As taught in the lecture, the KDE does not depend on the no. of samples but only on the kernel. We can quite confirm that by plotting the KDE for $N = 5000$ samples.

2. Although there are a few changes due to the increased no. of samples.
3. All the plots have comparitavely smoother peaks than earlier. We suspect that this could be due to the fact that their are more samples that means their are more no. of kernels summing over and creating more smoother versions.
4. But even with this the most optimal choice for bandwidth remains same as there is still noise and other problems such as over and undersmoothing.

```
[ ]: fig,ax = plt.subplots(1,2,figsize=(18,6))

     #plot1
     x = np.linspace(-10,10,1000)
     w = [0.1,0.5,1,3,5]
     [ax[0].plot(x,kernels(x,0,wi).biweight()/np.max(kernels(x,0,wi).
      ↪biweight()),label=f'k(w={wi})') for wi in w]

     ax[0].set_xlabel('x')
     ax[0].set_ylabel('k')
     ax[0].set_title(r'Normalised Quartic Kernel for $\mu=0$ and $w=1$')
     ax[0].legend(loc=(1.02,0.76))
     # ax[0].set_aspect(1)
     ax[0].grid()

     #plot2
     x1 = np.linspace(-2.5,2.5,1000)
     w1 = np.arange(1,2,0.2)
     [ax[1].plot(x1,kernels(x1,0,wi).biweight()/np.max(kernels(x1,0,wi).
      ↪biweight()),label=f'k(w={wi:.2f})') for wi in w1]

     # ax[1].set_xlim(-1.5,1.5)
     # ax[1].set_ylim(-0.01,0.1)
     ax[1].set_xlabel('x')
     ax[1].set_ylabel('k')
     ax[1].set_title(r'Normalised Quartic Kernel for $\mu=0$ and $w=1$')
     ax[1].legend(loc=(1.02,0.76))
     # ax.set_aspect(1)
     ax[1].grid()

     fig.tight_layout()
```

```
/var/folders/zd/rsl4z8j50cg9m4ll966fsyn80000gn/T/ipykernel_13565/777831253.py:6:
RuntimeWarning: invalid value encountered in divide
  [ax[0].plot(x,kernels(x,0,wi).biweight()/np.max(kernels(x,0,wi).biweight()),la
bel=f'k(w={wi})') for wi in w]
```





##

Problem 3: Mean-Shift

[ ]:
```
from IPython.display import Image
Image(filename='ex3aMS.jpg')
```

[ ]:

Exercise 3A

Update step with Epanechnikov kernel looks like:-

$$x_j^{t+1} = x_j^t + \alpha_j^t \frac{2}{n} \sum_{i:\|x_i - x_j^t\|<1} (x_i - x_j^t)$$

we want the update step to be equal to mean shift

i.e

$$x_j^{t+1} - x_j^t = \text{mean shift}$$

$$x_j^{t+1} - x_j^t = m(x_i) - x_i$$

where
$$m(x_i) - x_i = \frac{\sum_j K(x_i, x_j^t)\, x_j^t}{\sum K(x_i, x_j^t)} - x_i$$

Therefore,

$$\alpha_j^t \frac{2}{n} \sum_{i:\|x_i - x_j^t\|<1} (x_i - x_j^t) = \frac{\sum_j K(x_i, x_j^t)\, x_j^t}{\sum K(x_i, x_j^t)} - x_i$$

$$\alpha_j^t = \frac{n}{2}\left[\frac{\sum_j K(x_i, x_j^t)\, x_j^t - x_i \sum K(x_i, x_j^t)}{\sum K(x_i, x_j^t) \sum_{i:\|x_i - x_j^t\|<1} (x_i - x_j^t)} \right]$$

As pointed out in the lecture, rather than taking step in the direction of local mean, we can directly move the point of interest to local mean. This is a sensible decision as it will make the algorithm fast, avoid getting stuck in smaller spurious clusters as well as avoid overshooting. This rate will also enable the data points to move faster when they are away from cluster peak and slower as they approach the cluster peak.

1.5 3(b) bonus

```
[ ]: # TODO: implement the update to the local mean

     def mean_shift_step(x, xt, r=1):
         """
         A single step of mean shift, moving every point in xt to the local mean of␣
      ↪points in x within a radious of r.

         Parameters
         ----------
         x : np.ndarray
             Array of points underlying the KDE, shape (d, N1)
         xt : np.ndarray
             Current state of the mean shift algorithm, shape (d, N2)
         n_components : int, optional
             Number of requested components. By default returns all components.

         Returns
         -------
         np.ndarray
             the points after the mean-shift step
         """
         # NOTE: For the excercise you only need to implement this for d == 1.
         #       If you want some extra numpy-practice, implement it for arbitrary␣
      ↪dimension

         assert xt.shape[0] == x.shape[0], f'Shape mismatch: {x.shape[0]}!={xt.
      ↪shape[0]}'

         # TODO: compute a N by N matrix 'dist' of distances,
         #       such that dists[i, j] is the distance between x[i] and xt[j]
         d=np.zeros((len(x),len(xt)))
         for i in range(0,len(xt)):
             for j in range(0,len(x)):
                 d[i,j] = np.sqrt(x[i]**2 + xt[j]**2)


         # TODO: threshold the distances with r to get an array of masks for every␣
      ↪data point
         masks = []
         for i in range(0,len(x)):
             ind = np.where(d[i]<r)[0]
             masks.append(d[i,ind])
```

```
    # TODO: compute the number of points in x within radius r of each xt
    counts = []
    for i in range(0,len(x)):
        counts.append(len(masks[i]))

    # TODO: compute the local means by summing over the neighbors of each␣
 ↪element in xt
    #       and dividing by the number of neighbors
    local_means = []
    local_means = []
    for i in range(0,len(masks)):
        if(counts[i]!=0):
            s = np.sum(masks[i])
            local_means.append(s/counts[i])
        else:
            local_means.append(0)
    return local_means
```

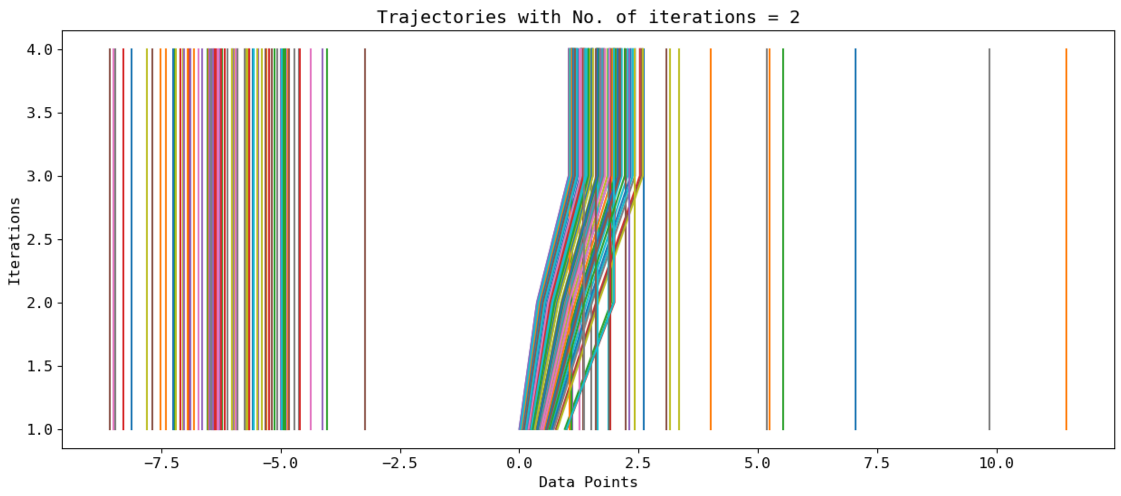
Directional mean shift implementation:

```
[ ]: # load the data
data = np.load("data/samples.npy")
x = data[:200]  # use e.g. the first 200 points
xt = x

trajectories = [xt]
max_steps = 100
for step in range(max_steps):

    # TODO: update xt with your mean shift step
    local_means = mean_shift_step(x, xt)
    xt  = xt+local_means

    # print(step)
    trajectories.append(xt)
    if np.allclose(trajectories[-1], trajectories[-2]):  # break in case of␣
 ↪convergence
        break
trajectories = np.stack(trajectories)
n_steps = len(trajectories) - 1
```

```
[ ]: #creating a step x 200 array to plot corresponding iteration number for each of␣
 ↪the array. We will use transpose of iter. iter.
#iter.T will have 200 rows of [1,...,step]
a = np.ones((1,x.shape[0]))
```

```
a = a.flatten()
iter = [a]
for i in range(0,trajectories.shape[0]-1):
    iter.append(a+1+i)
iter = np.array(iter)


# TODO: plot the trajectories
plt.figure(figsize=(15,6))
for i in range(0,x.shape[0]):
    plt.plot(trajectories.T[i],iter.T[i]);
plt.xlabel("Data Points")
plt.ylabel("Iterations")
plt.title(f"Trajectories with No. of iterations = {step}")
```

```
[ ]: Text(0.5, 1.0, 'Trajectories with No. of iterations = 2')
```



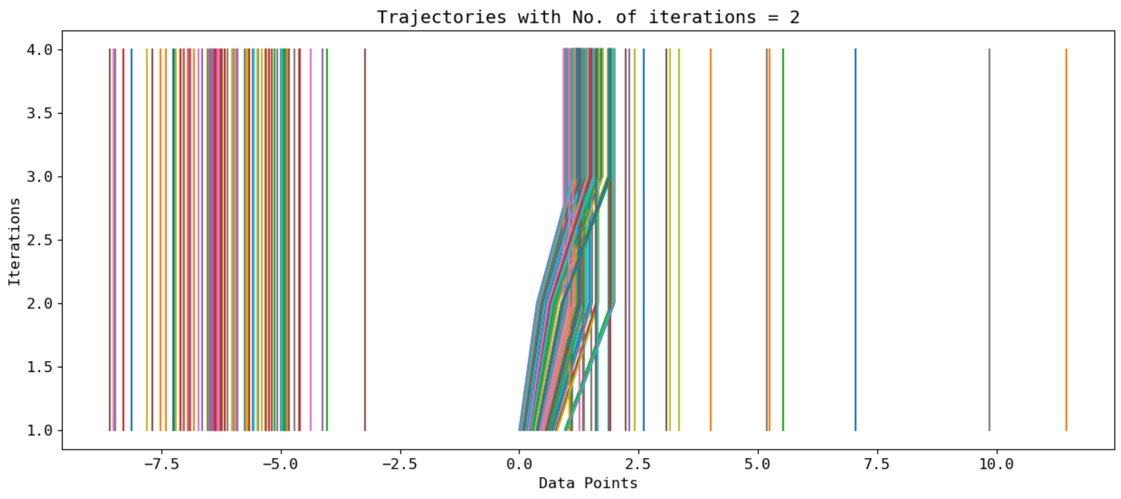
"Blurring" mean shift implementation:

```
[ ]: # TODO: repeat the above for "blurring" mean shift

x = data[:200]  # use e.g. the first 200 points
xt = x
trajectories = [xt]
max_steps = 100
for step in range(max_steps):

    # TODO: update xt with your mean shift step
    local_means = mean_shift_step(xt, xt) #we reuse the shifted datapoints␣
 ↪instead of the originaal data points.
```

```
    xt  = xt+local_means

    # print(step)
    trajectories.append(xt)
    if np.allclose(trajectories[-1], trajectories[-2]):  # break in case of␣
↪convergence
        break
trajectories = np.stack(trajectories)
n_steps = len(trajectories) - 1
```

[ ]:
```
a = np.ones((1,x.shape[0]))
a = a.flatten()
iter = [a]
for i in range(0,trajectories.shape[0]-1):
    iter.append(a+1+i)
iter = np.array(iter)

plt.figure(figsize=(15,6))
for i in range(0,x.shape[0]):
    plt.plot(trajectories.T[i],iter.T[i]);
plt.xlabel("Data Points")
plt.ylabel("Iterations")
plt.title(f"Trajectories with No. of iterations = {step}")
```

[ ]:
```
Text(0.5, 1.0, 'Trajectories with No. of iterations = 2')
```

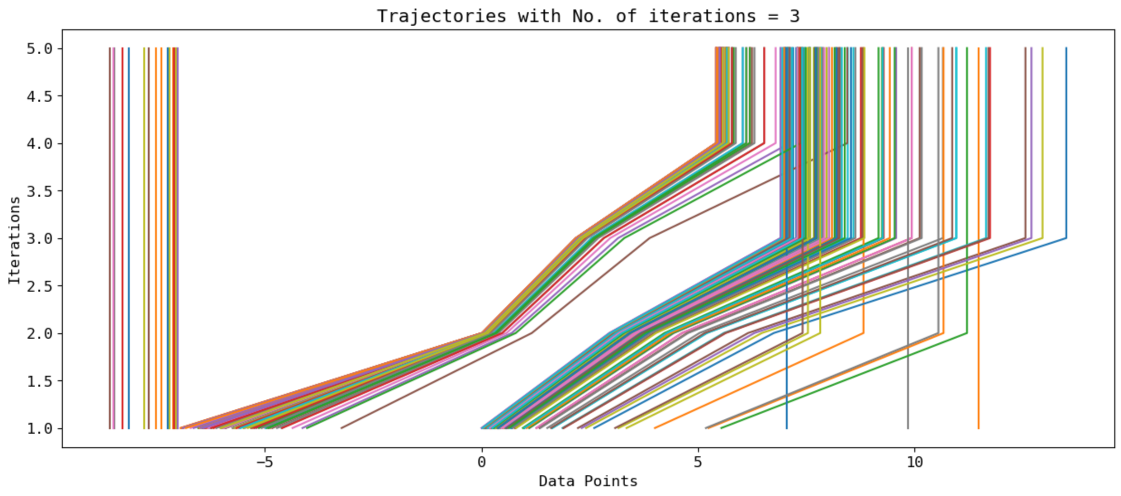


1.5.1 The blurring mean-shift is an accelerated version which uses the original data only in the first step, then re-smoothes previous estimates. We see that the algorithm has converged in 2 iterations where as the previous one had converged in 2 iterations.

Comments: We do not understand why the points lying under first KDE peak (as seen in Ex1) did not move from their positions. One explaination could be that the $r = 1$ was small and therefore couldn't encompass all the samples. Therefore, to confirm this, we ran mean shift again but with $r = 7$.

```
[ ]: x = data[:200]  # use e.g. the first 200 points
     xt = x
     trajectories = [xt]
     max_steps = 100
     for step in range(max_steps):

         # TODO: update xt with your mean shift step
         local_means = mean_shift_step(xt, xt,r=7) #we reuse the shifted datapoints␣
      ↪instead of the originaal data points.
         xt   = xt+local_means

         # print(step)
         trajectories.append(xt)
         if np.allclose(trajectories[-1], trajectories[-2]):  # break in case of␣
      ↪convergence
             break
     trajectories = np.stack(trajectories)
     n_steps = len(trajectories) - 1

     a = np.ones((1,x.shape[0]))
     a = a.flatten()
     iter = [a]
     for i in range(0,trajectories.shape[0]-1):
         iter.append(a+1+i)
     iter = np.array(iter)

     plt.figure(figsize=(15,6))
     for i in range(0,x.shape[0]):
         plt.plot(trajectories.T[i],iter.T[i]);
     plt.xlabel("Data Points")
     plt.ylabel("Iterations")
     plt.title(f"Trajectories with No. of iterations = {step}")
```

```
[ ]: Text(0.5, 1.0, 'Trajectories with No. of iterations = 3')
```



In order to test, we repeated the algorithm with $r = 7$. We suspect the points to the left seem to join the bigger cluster instead of clustering amongst themselves.

##

Problem 4(a)

[ ]:
```
from IPython.display import Image
Image(filename='4a.jpg')
```

[ ]:

4.(i) → $\mathcal{E}(C, M; K) = \sum_{n=1}^{N} \sum_{K=1}^{K} m_{KN} \|x_n - c_k\|^2$

update for m_{KN}. (Keeping in Mind it either takes 1 or 0)

diff $\mathcal{E}$ w.r.t. 'm_{nk}'.

$$\frac{\partial \mathcal{E}}{\partial m_{nk}} = \sum_{n=1}^{N} \sum_{k=1}^{K} \|x_n - c_k\|^2$$

which doesn't give us much info about the particular value of 'm_{nk}'

thus we can minize it for a specific distance when $\|x_n - c_k\|^2$ is minimum

thus $m_{nk} = \begin{cases} 1 & \text{iff } k = \arg\min \|x_n - c_k\|^2 \\ 0 & \text{otherwise} \end{cases}$

update for c_k ⇒.

diff $\mathcal{E}$ w.r.t. c_k. we get

$$\frac{\partial \mathcal{E}}{\partial c_k} = 2 \sum_{n=1}^{N} m_{ik} (x_i - c_k) = 0$$

$$\boxed{c_k = \frac{\sum_{i=1}^{N} m_{ik} x_i}{\sum_{i=1}^{N} m_{ik}}}$$

this step recomputes the new centroid for each cluster.

1.6 4 K-Means (b)

```
[ ]: from sklearn.cluster import KMeans

     def kmeans_step(x, k, c=None, init='random'):
         """
         K-Means clustering on a p x N data matrix.

         Parameters
         ----------
         x : np.ndarray
             Data matrix of shape (p, N).
         k : int
             Number of cluster.
         c : np.ndarray, optional
             Current cluster centers. If None, the initialization as specified by␣
      ↪'init' will be used.
         init: str
             The initialization method to be used if c is None.

         Returns
         -------
         float, np.ndarray
             cost after the step, updated cluster centers
         """

         if c is not None:
             assert c.shape[1] == k


         # TODO: set n_cluster, init, n_init and max_iter appropriately
         single_step_kmeans = KMeans(
             n_clusters=k,
             init=init,
             max_iter=1,
             n_init=1,
         )

         # TODO call the single_step_kmeans
         ssk =single_step_kmeans.fit(x.T)  ## because data has to be in the shape␣
      ↪n_samples,n_features..

         # TODO get the current energy (you don't have to compute it - it's stored␣
      ↪in single_step_kmeans)
```

```
    E = ssk.inertia_
    # TODO read out cluster centers
    C =ssk.cluster_centers_

    return E, C
```

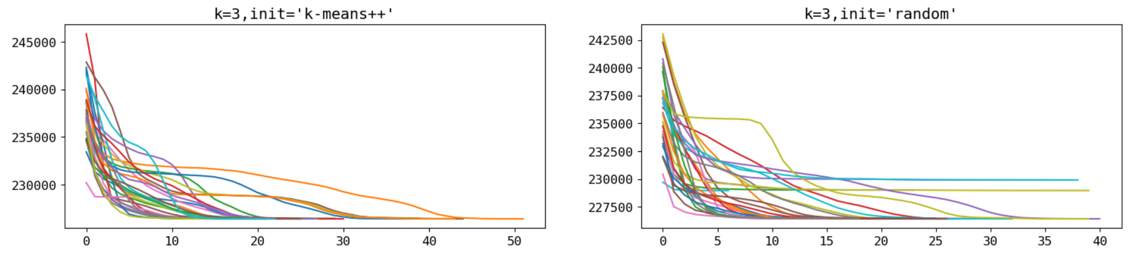
```
[ ]: # load the data (you can try both with the full normalized feature and the 2d␣
      ↪umap projection)

     features = np.load('data/dijet_features_normalized.npy')  # full features
     # features = np.load('data/dijet_features_umap.npy')      # umap projection

     print(f'{features.shape=}')
```

```
features.shape=(116, 2233)
```

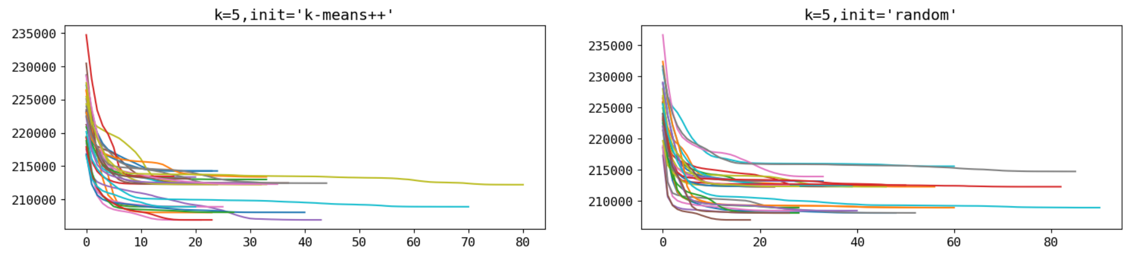
```
[ ]: def kmeans(x, k, max_steps=100, init='random'):
         c = None
         energies = []  # list of energies over the iterations.
         for i in range(max_steps):
             # Call the kmeans_step implemented above to get the energy and the next␣
      ↪cluster centers
             E, C = kmeans_step(x, k=k, c=c, init=init)
             energies.append(E)
             init = C  ##changeing the init after each step


             # Stop the loop if there was no improvement
             if i>=2 and energies[-1] == energies[-2]:
                 break
         energies = np.array(energies)
         return energies  # return array of energies

     for k in [3,5,10,20]:
         plt.figure(figsize = (18,16))
         plt.subplots_adjust(hspace=0.2)
         plt.suptitle(f"Resulting energy trajectories for k-means++ and random",␣
      ↪fontsize=18, y=0.95)
         plt.tight_layout()


         #setting no. of rows and columns for subplot
         ncols = 2
         nrows = 4

         for n,init in enumerate(['k-means++', 'random']):
```

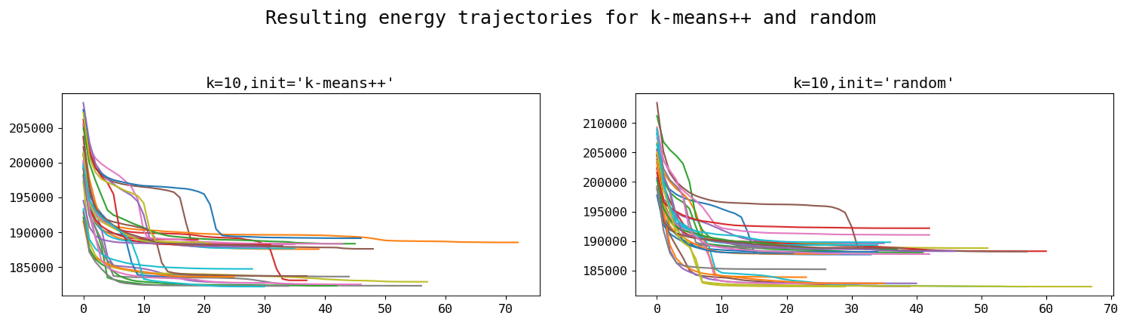
```
        # TODO: for the given k and init, run k-means 20 times or more (using␣
↪the kmeans function above)
        #       and plot the resulting energy trajectories

        #adding subplot iteratively
        ax = plt.subplot(nrows, ncols, n + 1)

        #plotting the data histogram and KDE
        for i in range(30):
            energy_arr = kmeans(features,k,max_steps = 100,init = init)
            ax.plot(np.arange(0,len(energy_arr)),energy_arr)

        # chart formatting
        ax.set_title(f'{k=},{init=}')
```

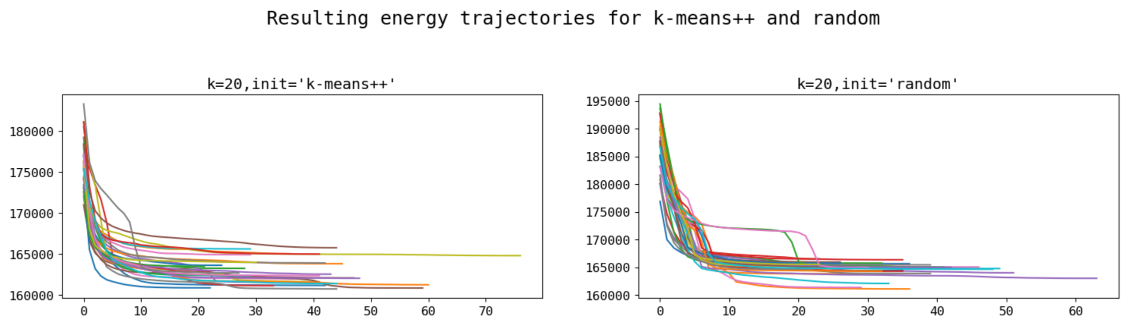
Resulting energy trajectories for k-means++ and random



Resulting energy trajectories for k-means++ and random







Interpretation of the result We can clearly see that the Energy converges to minimum faster when **K-means++** initialization is used than the **random** initialization.