Pattern Recognition

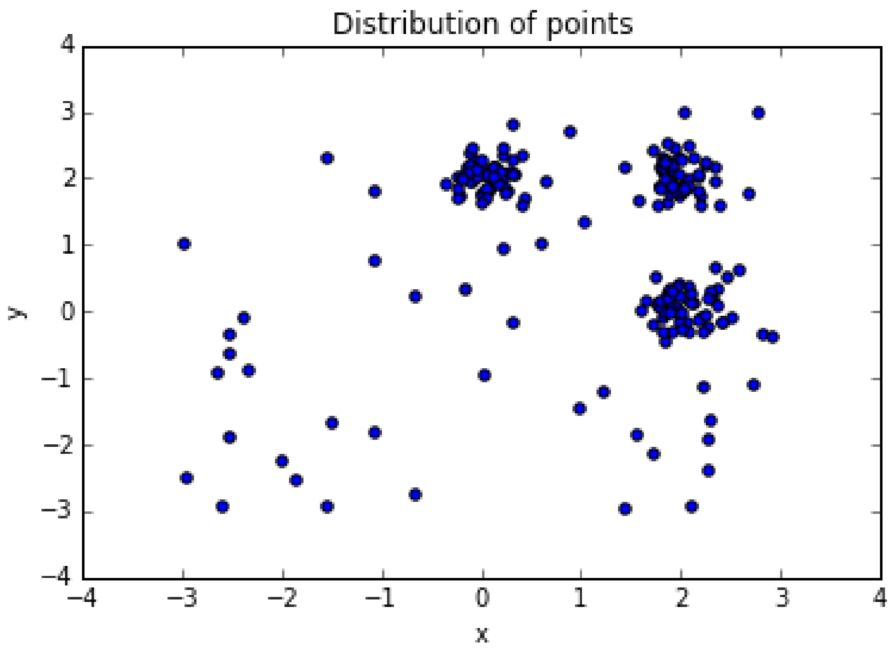
Project 3: Clustering and Dimensionality Reduction

Alina Arunova
Mohammad Ali Ghasemi
Aditya Kela
Aleksandr Korovin
Marina Mircheska
Sattar Rahimbeyli

Task 3.1

Implement the following algorithms:

1. Lloyd's algorithm for k-means clustering (e.g. simply using scipy)
2. Hartigan's algorithm for k-means clustering
3. MacQueen's algorithm for k-means clustering

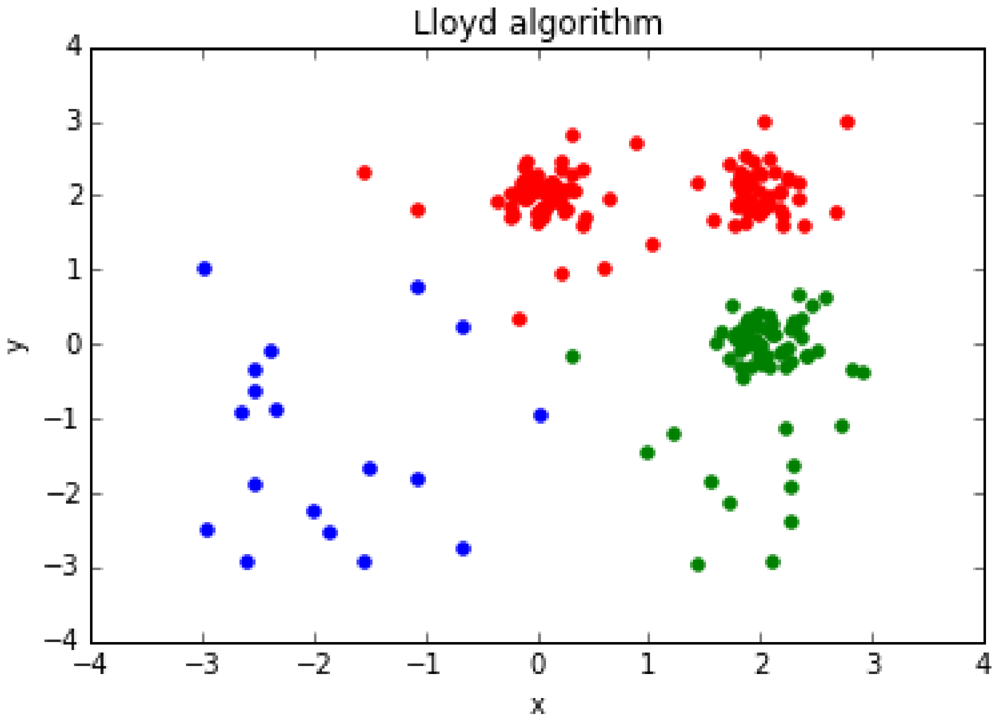


- For k = 3, run each algorithm on the above data and plot your results.
- Measure the run times of each of your implementations (run them each at least 10 times and determine their average run times).

Task 3.1:

Lloyd‘s algorithm

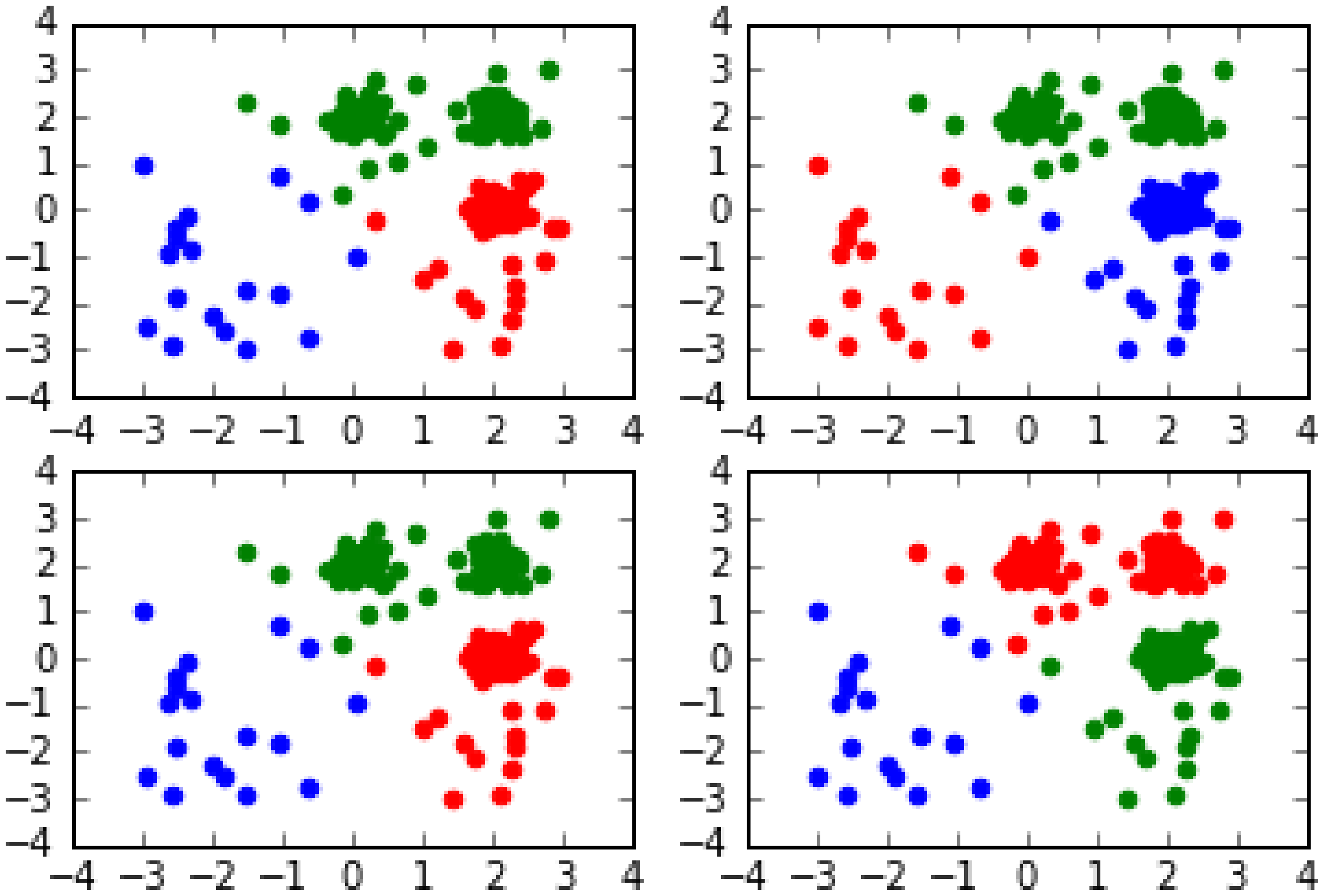
1. Choose the number of clusters
2. Choose the metric to use
3. Choose the method to pick initial centroids
4. Assign initial centroids
5. While metric(centroids, cases)>threshold
   a. For ***i*** <= ***nb*** cases
      - Assign case to closest cluster according to metric
   b. Recalculate centroids



Average time for Lloyd’s algorithm: 0.0138999938965

Task 3.1:

Lloyd‘s algorithm



Advantages:

- For large data sets
- Discrete data distribution
- Optimize total sum of squares

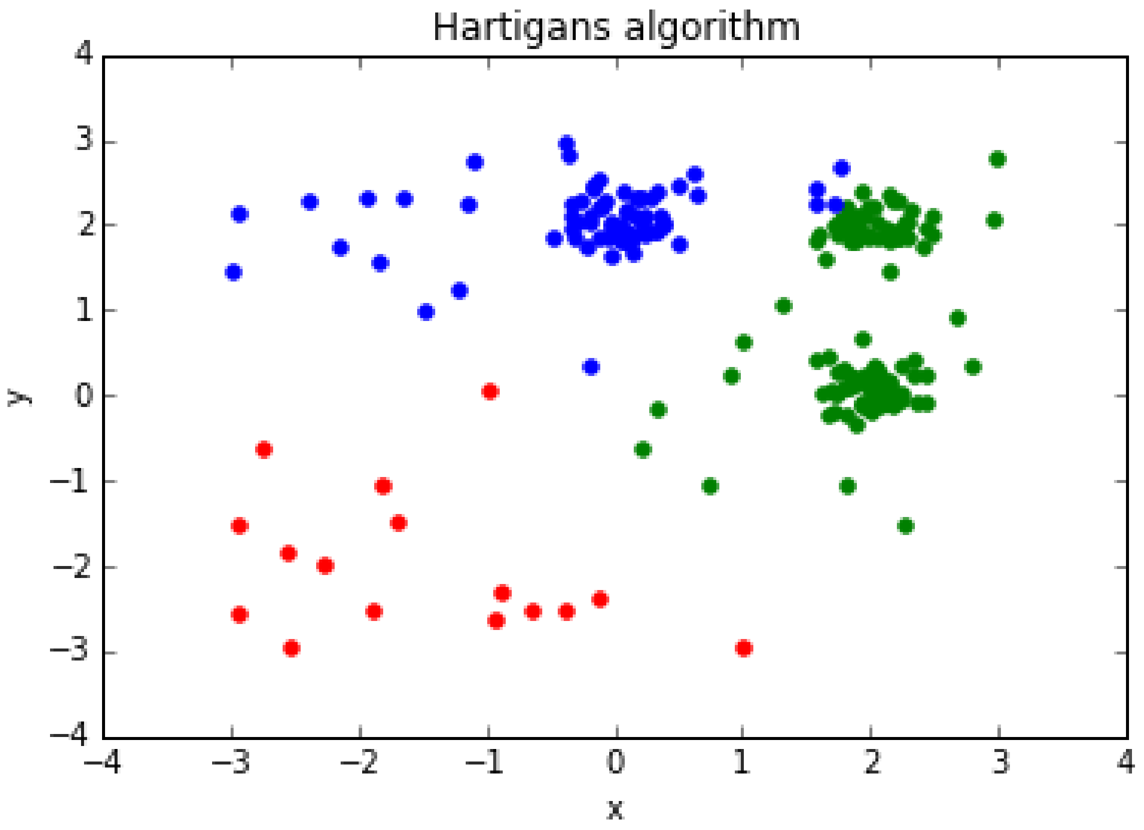
Disadvantages:

- Slower convergence
- Possible to create empty clusters

Task 3.1

Hartigan‘s algorithm

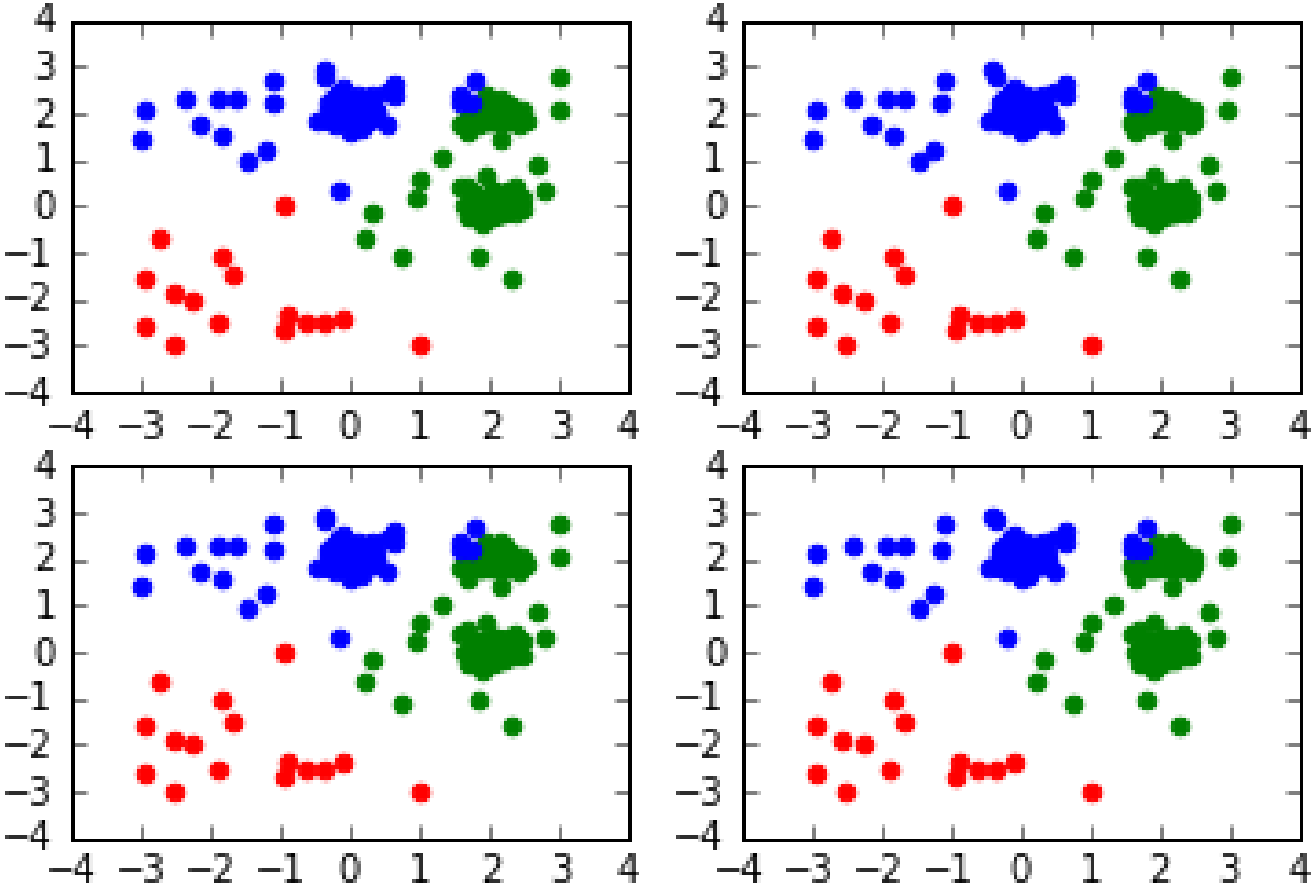
1. Choose the number of clusters
2. Choose the metric to use
3. Choose the method to pick initial centroids
4. Assign initial centroids
5. Assign cases to closest centroid
6. Calculate centroids
7. For ***j*** <= ***nb*** clusters, if centroid j was updated last iteration
   a. Calculate SSE within cluster
   b. For ***i*** <= ***nb*** cases in cluster
      I. Compute SSE for cluster ***k*** != ***j*** if case included
      II. If SSE cluster ***k*** < SSE cluster ***j***, case char cluster



Average time for Hartigan’s algorithm: 4.97979998589

Task 3.1

Hartigan‘s algorithm



Advantages:

- Fast initial convergence
- Optimize within-cluster sum of squares

Disadvantages:

- Need to store the two nearest-cluster computations for each case
- Sensitive to the order the algorithm is applied to the cases

Task 3.1

MacQeen‘s algorithm

for all $C_i \in \{C_1, \dots C_k\}$, initialize μ_1 and set $n_i = 0$

for all $x_j \in \{x_1, x_2 \dots\}$,

determine winner centroid

$$\mu_w = \operatorname{argmin}\left\|x_j - \mu_j\right\|^2$$

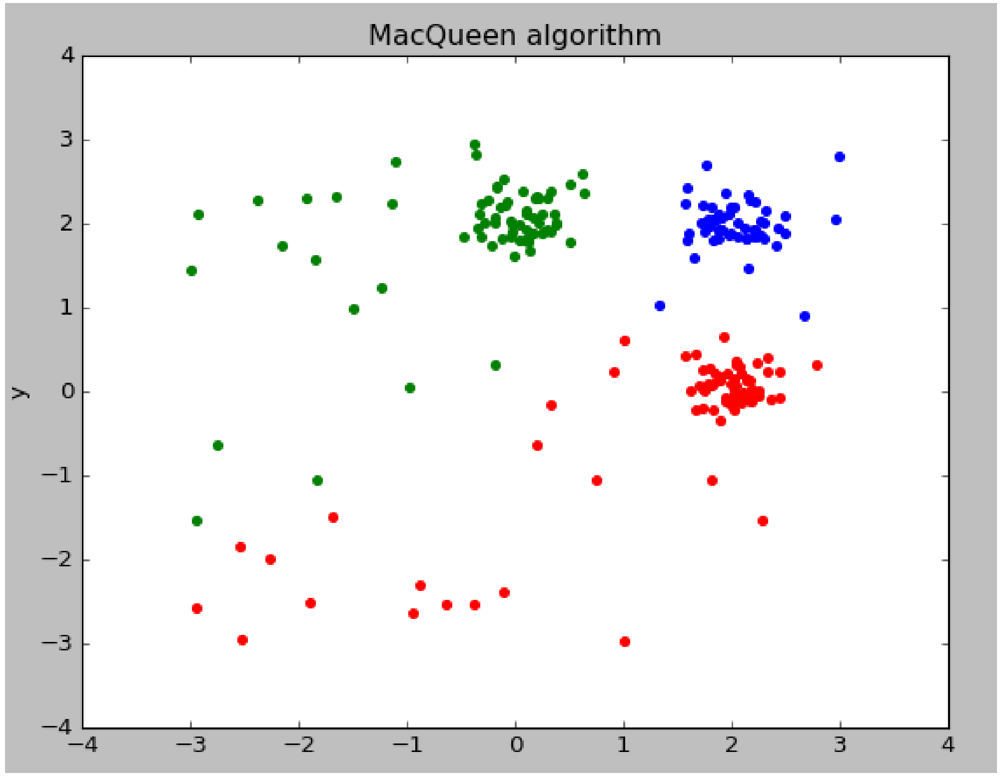
update cluster size and centroid

$$n_w \leftarrow n_w + 1$$

$$\mu_w \leftarrow \mu_w + \frac{1}{n_w}\left[x_j - \mu_w\right]$$

for all $C_i \in \{C_1, \dots C_k\}$

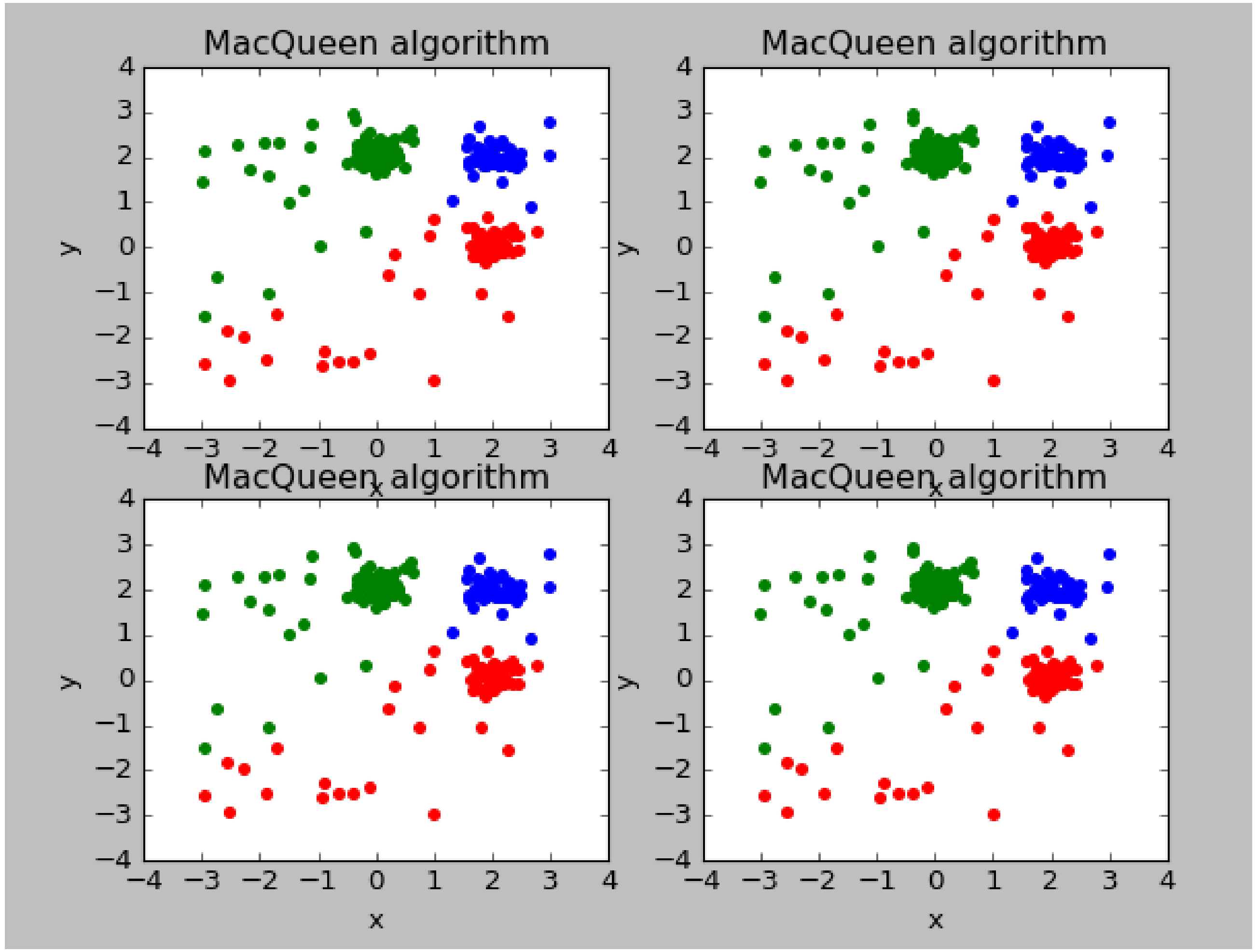
$C_i = \{x \in X \mid \|x - \mu_i\|^2 \le \|x - \mu_l\|^2\}$



Average time for MacQueen algorithm: 0.0348999977112

Task 3.1

MacQeen‘s algorithm



Advantages:

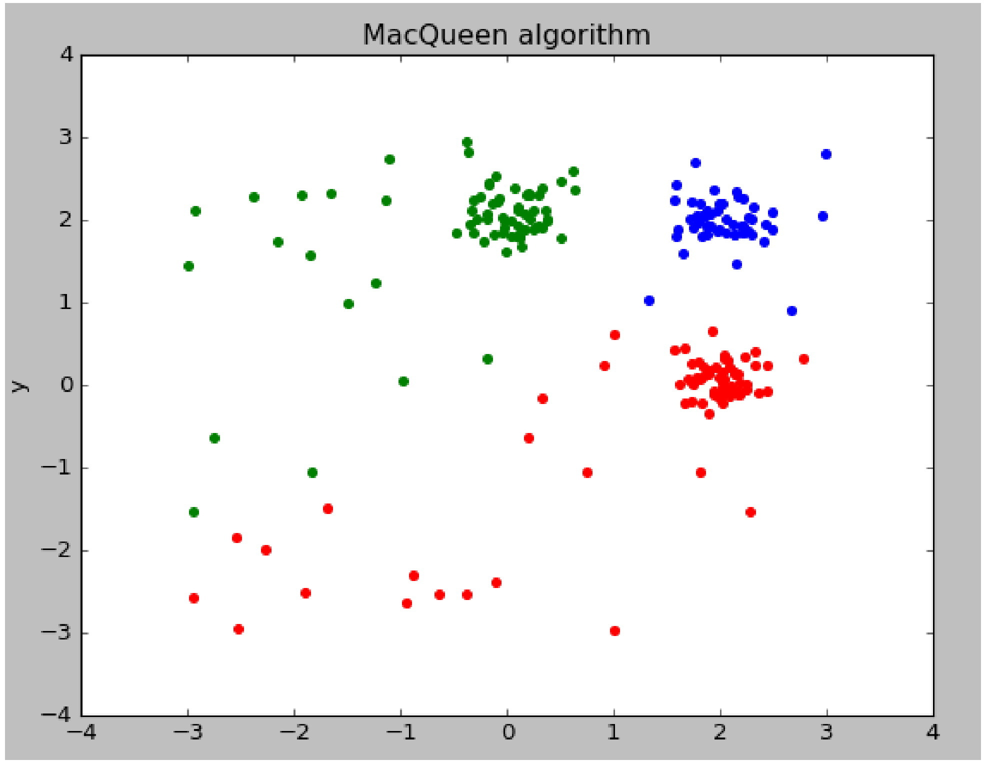
- Fast initial convergence
- Optimize total sum of squares

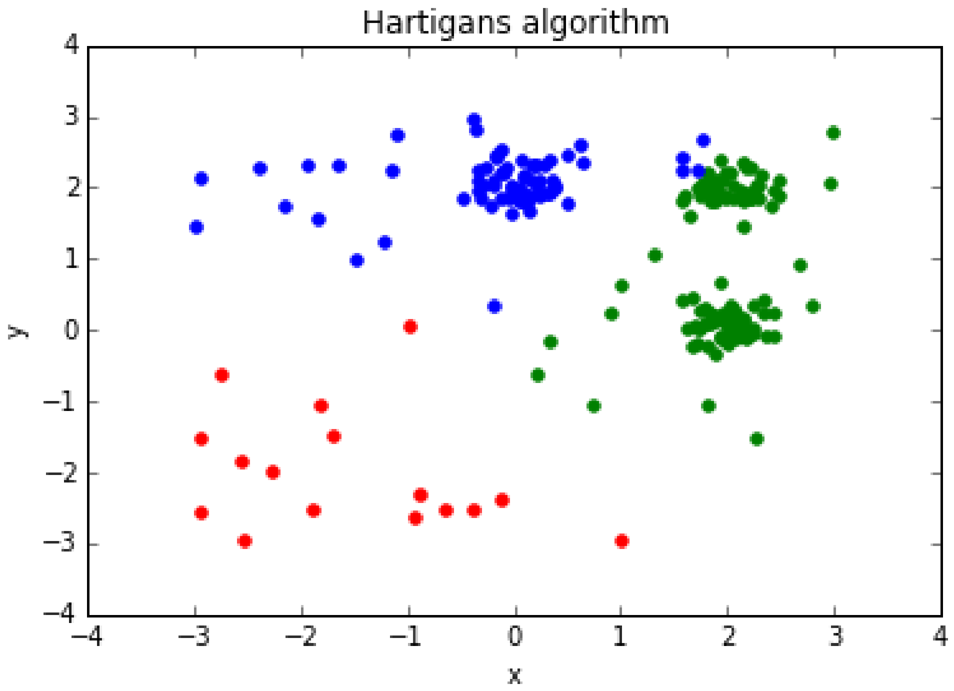
Disadvantages:

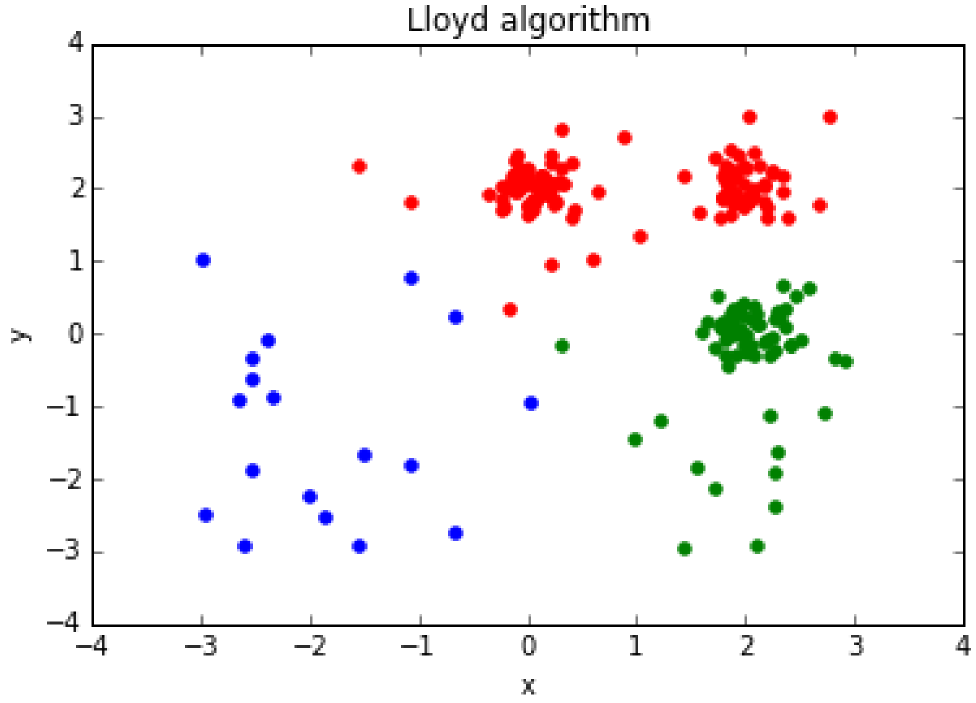
- Need to store the two nearest-cluster computations for each case
- Sensitive to the order the algorithm is applied to the cases

Task 3.1

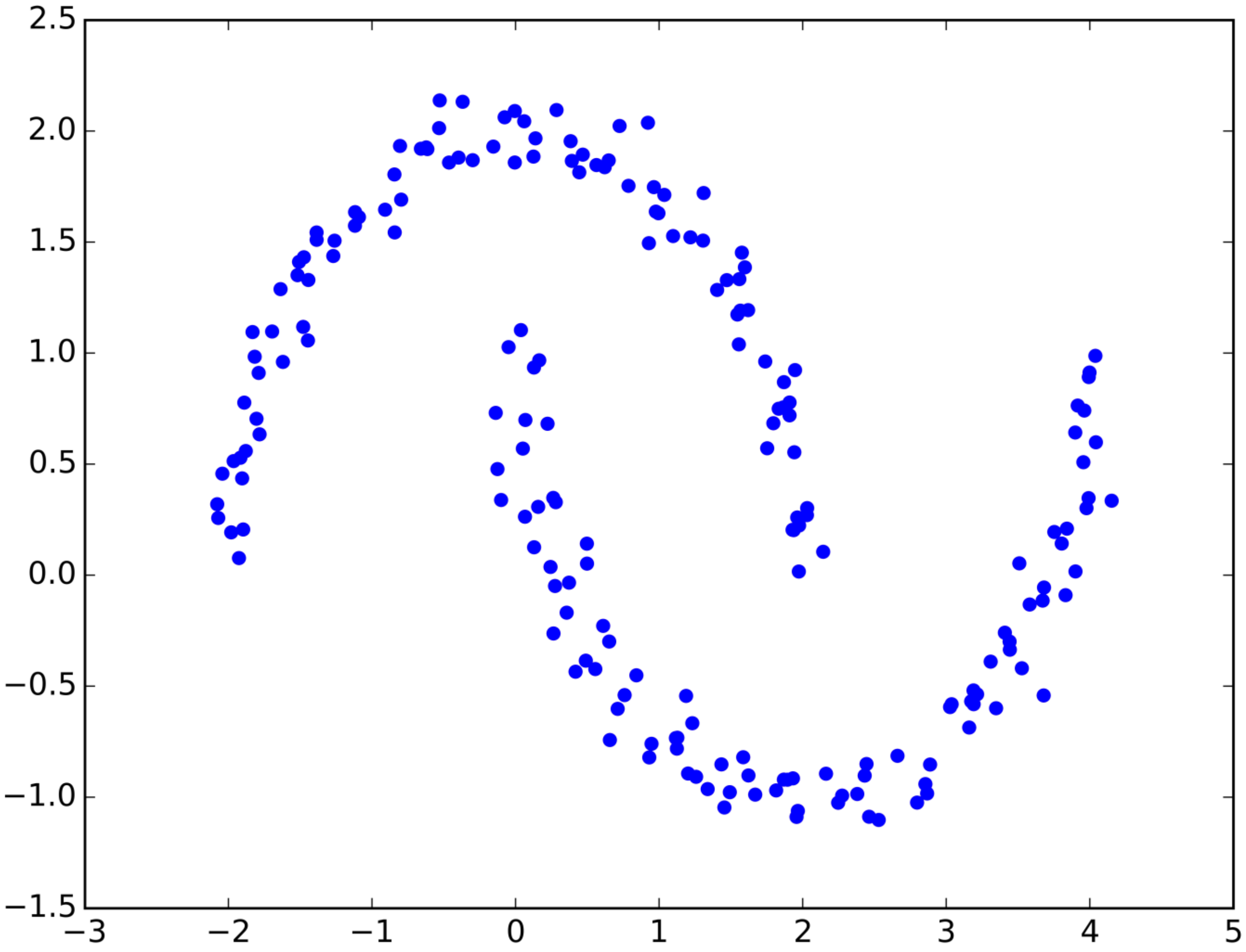
Algorithms







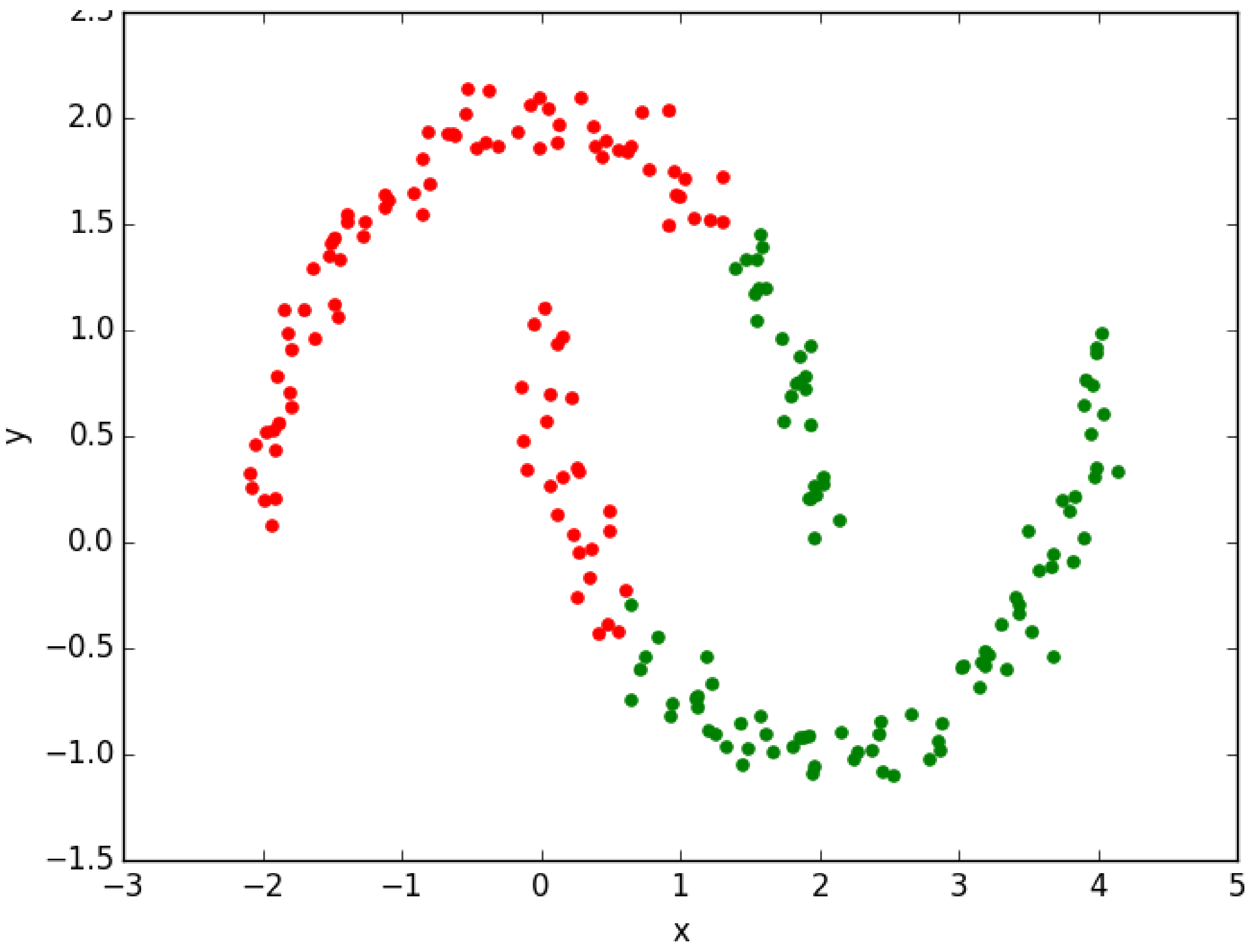
Task 3.2 Spectral Clustering



data in file data-clustering-2.csv

Task 3.2 Spectral Clustering

Results when applying MacQueen algorithm for k = 2



Task 3.2 Spectral Clustering

Idea behind Spectral Clustering:

Cluster data that is connected but not necessarily compact or clustered in convex boundaries

How?

Interpret the data as a graph and exploit graph properties to derive a clustering, i.e. use the spectrum of the similarity matrix of the data

Steps:

1. Define the affinity (similarity) matrix S as $S_{ij} = \exp(-\beta \|x_i - x_j\|^2)$ (Gaussian kernel similarity function)
2. Compute the Laplacian matrix $L = D - S$, where $D_{ij} = \begin{cases} \sum_j S_{ij} & if\ i = j \\ 0 & \end{cases}$
3. Compute the eigenvectors and eigenvalues of L
4. Use the Fiedler vector to cluster the points

Task 3.2 Spectral Clustering

Solution: code

Create the similarity matrix S:

```
for i in range(S.shape[0]):
    for j in range(S.shape[1]):
        S[i][j] = np.exp(-beta*np.linalg.norm(data[i]-data[j])**2)
```

Compute the Laplacian matrix:

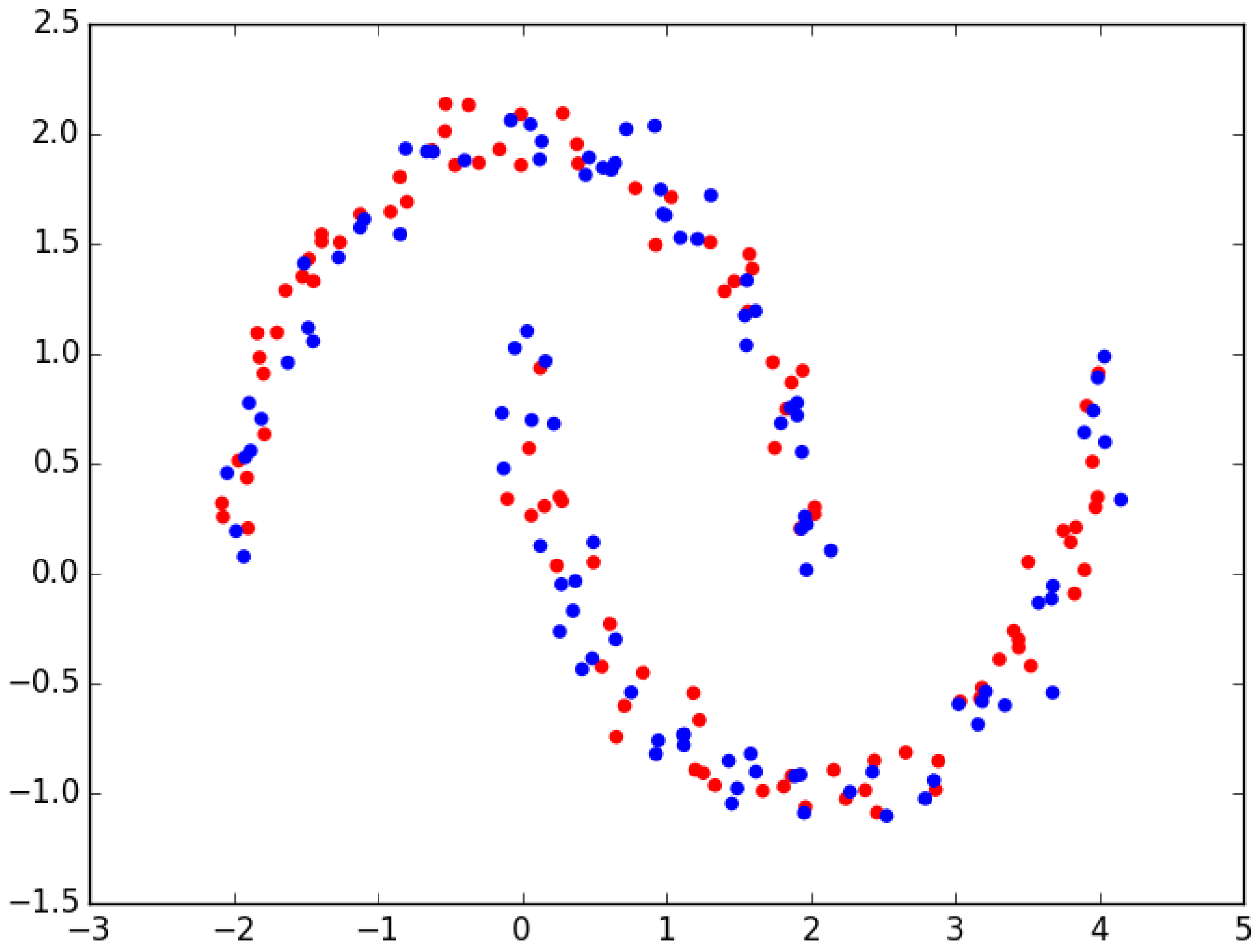
```
L = csgraph.laplacian(S, normed=False)
```

Compute the eigenvalues and eigenvectors of L and extract the Fielder vector :

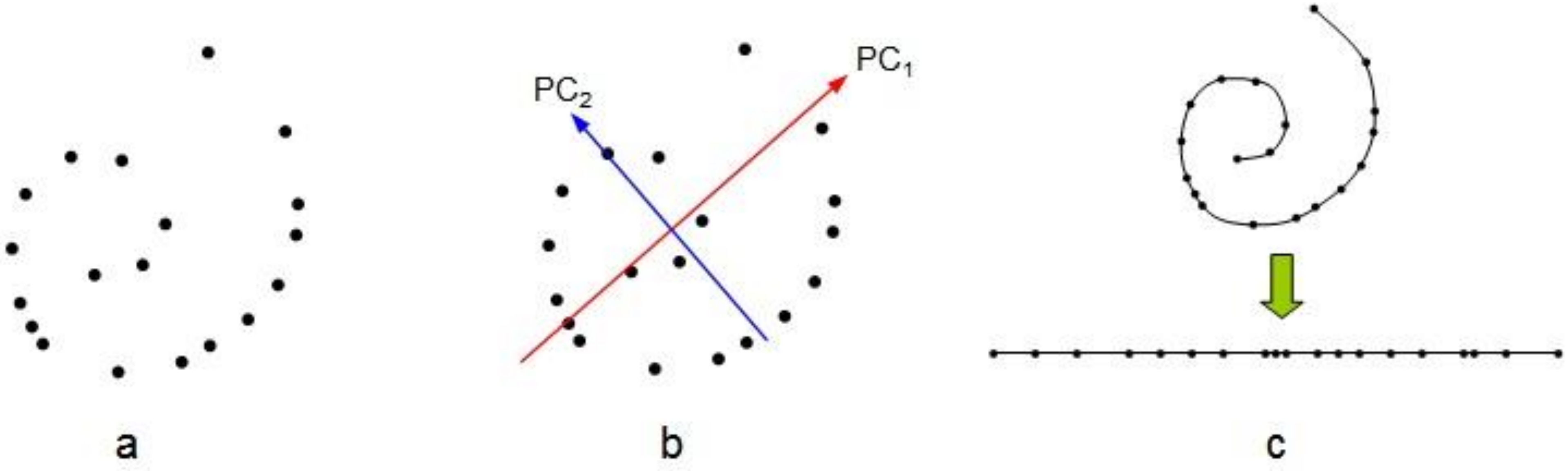
```
eigenvalues, eigenvectors = np.linalg.eig(L)

dict = {}
for i in range(a.shape[0]):
    dict[eigenvalues[i]] = eigenvectors[i]

t = sorted(dict.items())
fiedler = t[1]
```

Task 3.2 Spectral Clustering

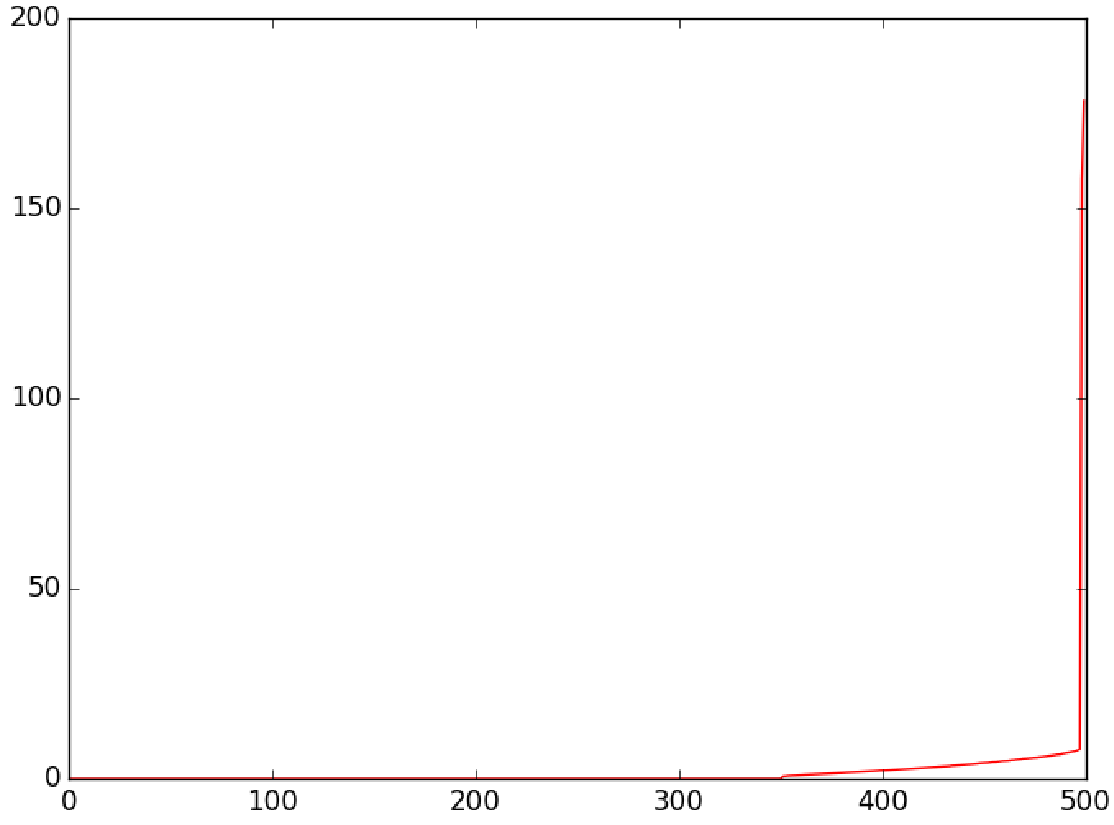
Resulting plot



Task 3.3: Dimensionality Reduction

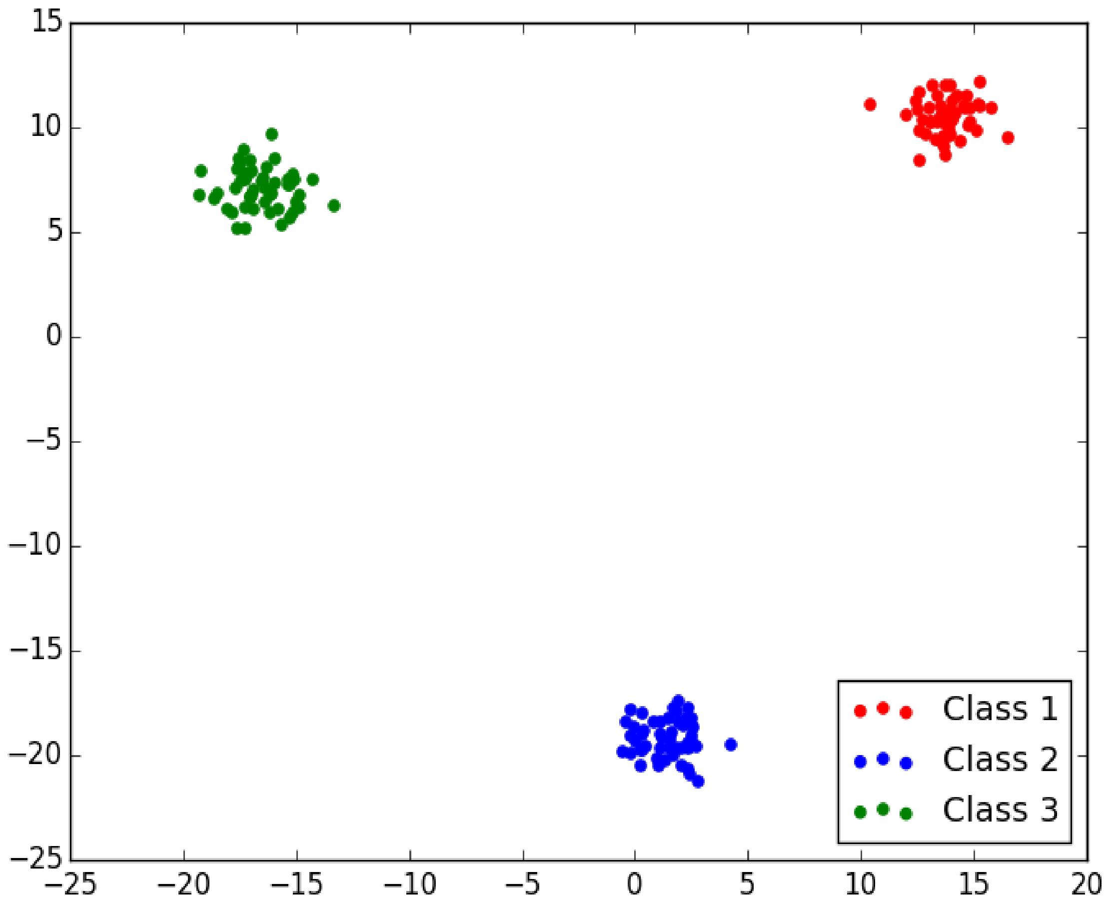


Task 3.3: Dimensionality Reduction



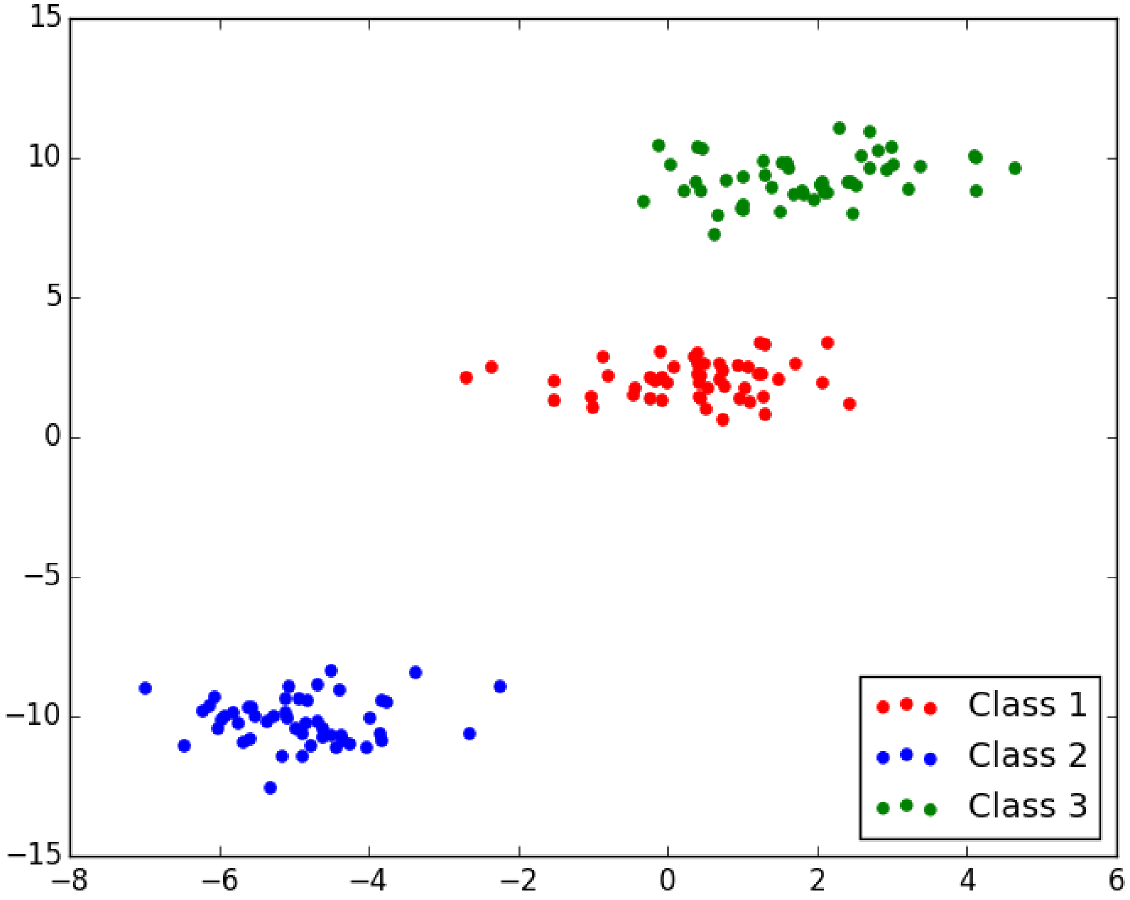
Eigenvalue Spectrum PCA

Task 3.3: Dimensionality Reduction



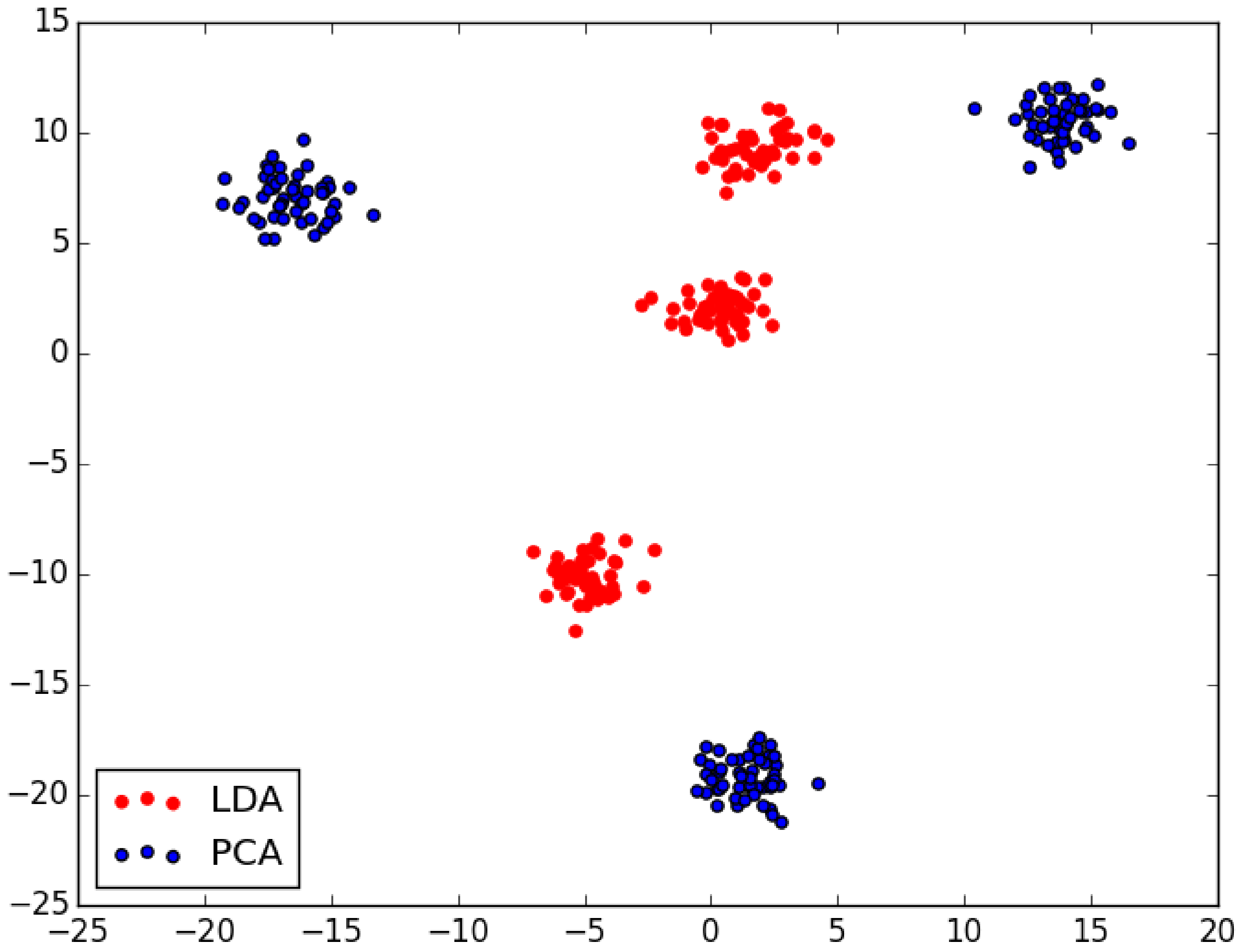
Dimensionality Reduction via PCA

Task 3.3: Dimensionality Reduction

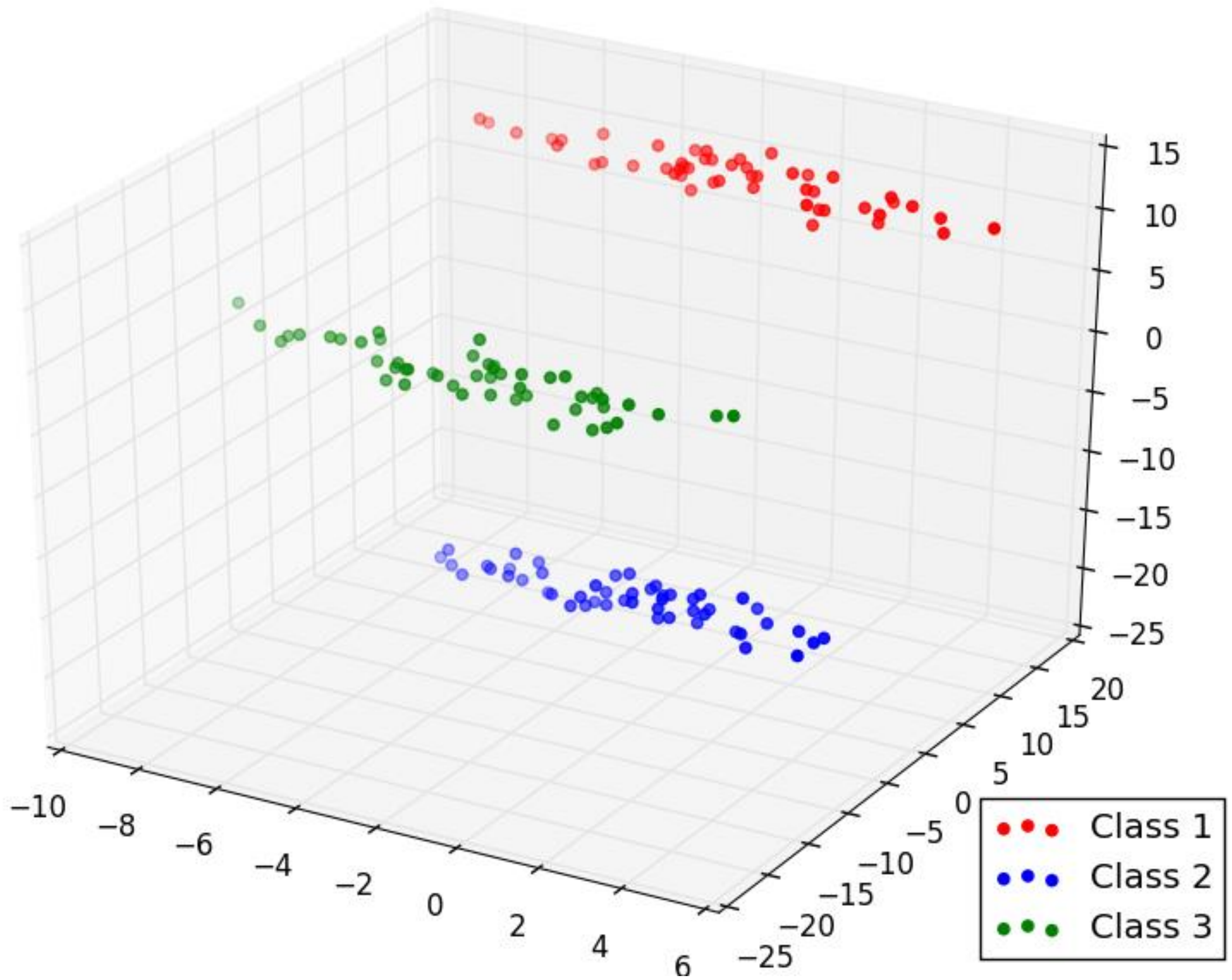


Dimensional Reduction via LDA

Task 3.3: Dimensionality Reduction

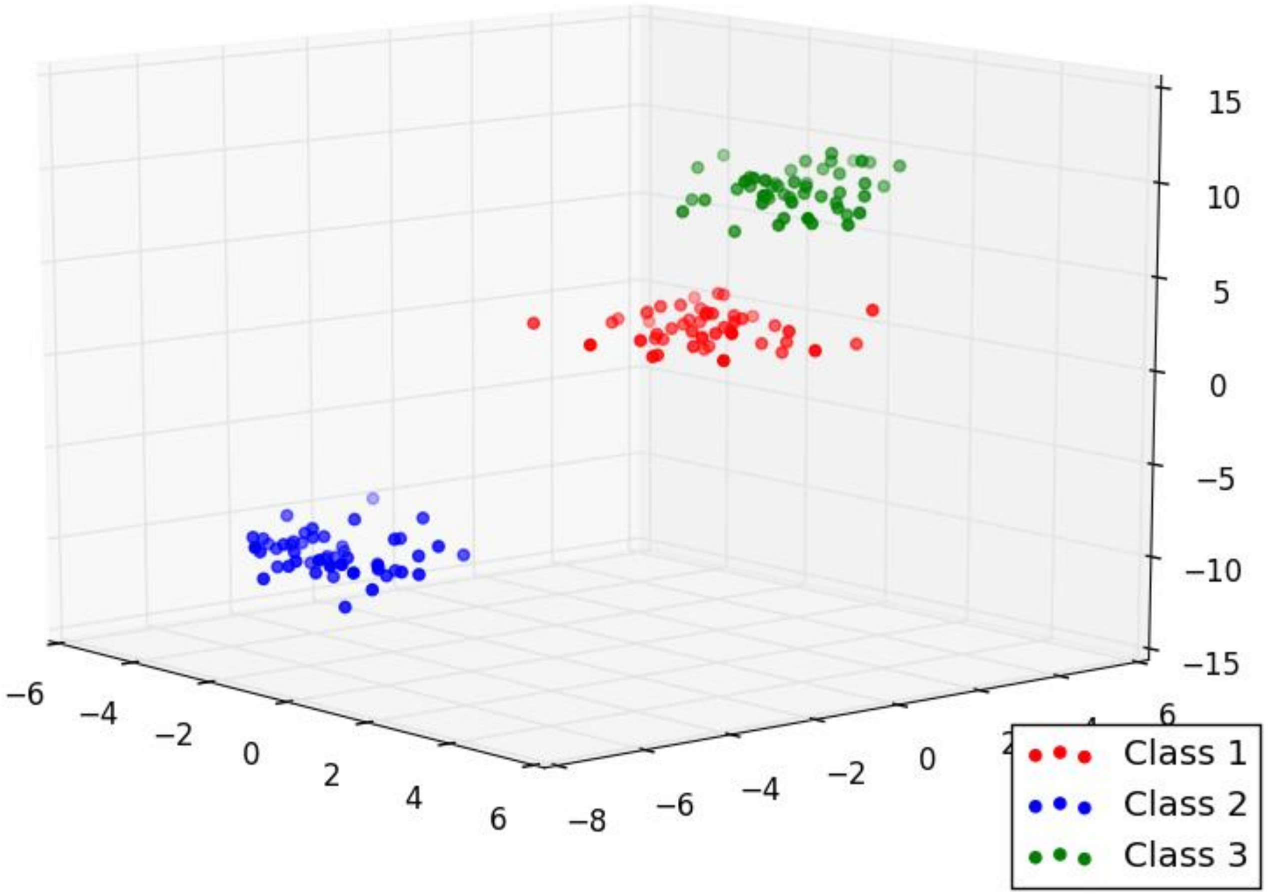


Task 3.3: Dimensionality Reduction



Dimensionality Reduction to 3D: PCA

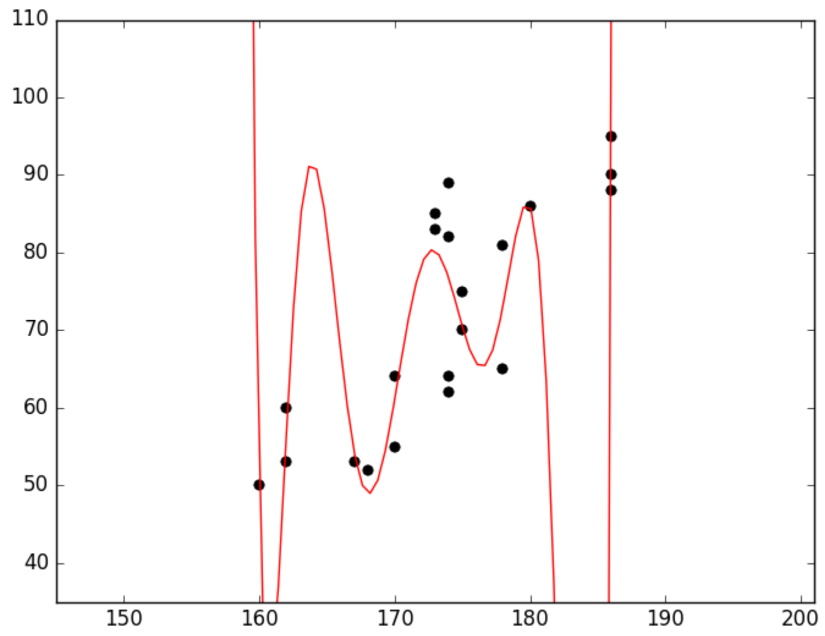
Task 3.3: Dimensionality Reduction



Dimensionality Reduction to 3D: PCA

Task 3.4: Exploring Numerical Instabilities

Method 1: In this method we use poly.polyfit function to derive coefficients.
It is clear that the outputs with methods 1 and 4 are the same in shapes.
The important thing which is most probable is that polynomial.polynomial.polyfit function is conditional.
On the other hand, polyfit issues a RankWarning when the least squares fit is badly conditioned .
This implies that the best fit is not welldefined due to numerical error.
The results may be improved by lowering the polynomial degree.



Task 3.4: Numerical Problmes which is derived

According the *y output* we have derived we faced with lots of numerical problems in M1 and M4.

This outputs are irrelevant and incorrect according something we expect and the function raise an error!

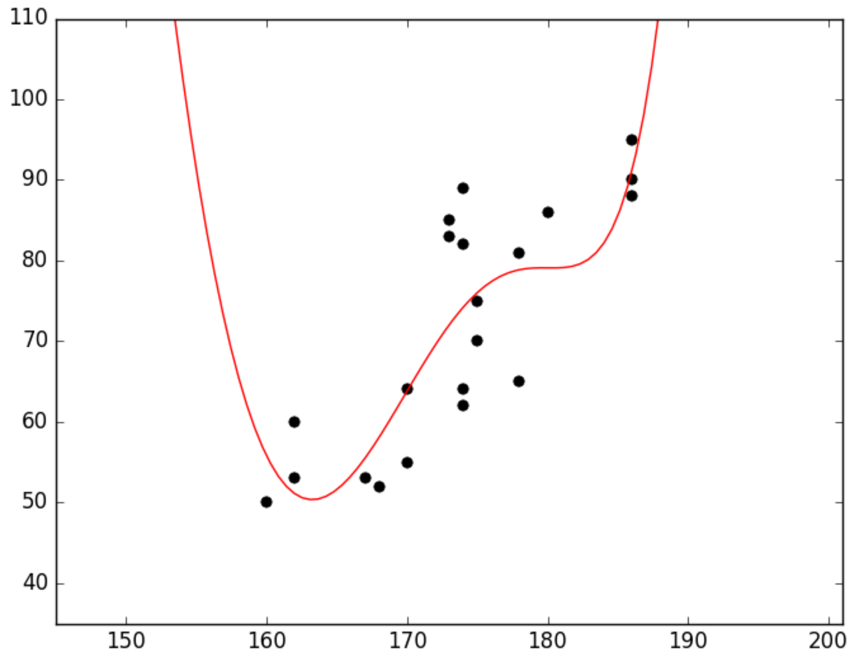
8.19608167e+05	6.81670567e+05	5.64054698e+05	4.64202275e+05
3.79820909e+05	3.08862510e+05	2.49502895e+05	2.00122615e+05
1.59288956e+05	1.25739065e+05	9.83641555e+04	7.61947679e+04
5.83870380e+04	4.42099373e+04	3.30334435e+04	2.43175867e+04
1.76023693e+04	1.24984816e+04	8.67880127e+03	5.87061902e+03
3.84857190e+03	2.42823621e+03	1.46033386e+03	8.25542847e+02
4.29838013e+02	2.00371094e+02	8.18153076e+01	3.31628418e+01
2.49404297e+01	3.67973633e+01	5.54575195e+01	7.29804688e+01
8.53103027e+01	9.10939941e+01	9.07408447e+01	8.56533203e+01
7.76776123e+01	6.87005615e+01	6.03491211e+01	5.38623047e+01
4.99832764e+01	4.89757080e+01	5.06700439e+01	5.45531006e+01
5.98830566e+01	6.57949219e+01	7.14361572e+01	7.60623779e+01
7.91262207e+01	8.03405762e+01	7.96998291e+01	7.74877930e+01
7.42302246e+01	7.06356201e+01	6.74954834e+01	6.55679932e+01
6.54487305e+01	6.74284668e+01	7.13912354e+01	7.66840820e+01
8.20941162e+01	8.58354492e+01	8.56463623e+01	7.90083008e+01
6.34854736e+01	3.72314453e+01	2.61596680e01	4.72154541e+01
9.82978516e+01	1.43082397e+02	1.64043335e+02	1.34238770e+02
1.44466553e+01	2.50219360e+02	7.34260498e+02	1.53640955e+03

```python
# Solve the least squares problem.
c, resids, rank, s = la.lstsq(lhs.T/scl, rhs.T, rcond)
c = (c.T/scl).T

# warn on rank reduction
if rank != order and not full:
    msg = "The fit may be poorly conditioned"
    warnings.warn(msg, pu.RankWarning)

if full:
    return c, [resids, rank, s, rcond]
else:
    return c
```

Task 3.4: Methods which are properly fitted to data

The infrastructure in Methods 2 and 3 are exactly the same and it doesn't have limit for defining higher degrees for polynomial function. That is why we do not have numerical problems after fitting a polynomial function with 10th degree on our trained data:

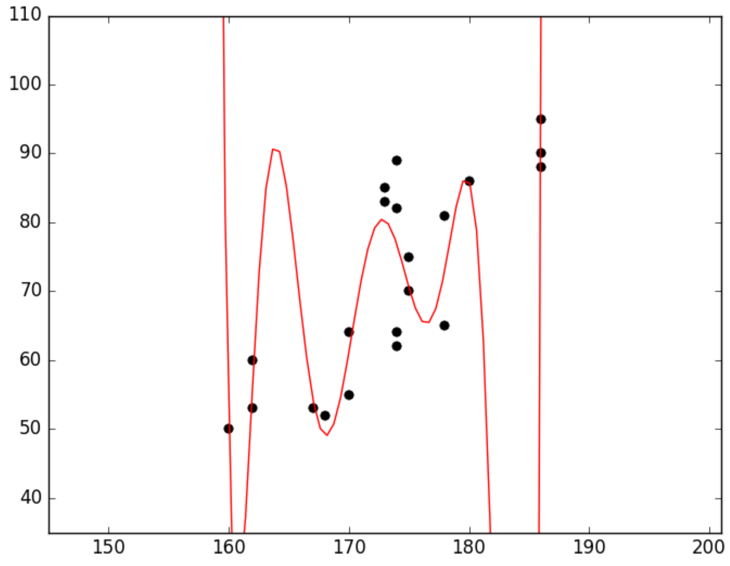
284.04199009	268.83083096	254.0957663	239.84845755	226.09956967
212.8587368	200.13452928	187.93442235	176.26476656	165.13076009
154.53642313	144.48457442	134.97681022	126.01348576	117.5936994
109.71527972	102.37477563	95.56744971	89.28727506	83.52693577
78.27783119	73.5300843	69.27255439	65.49285411	62.17737142
59.31129627	56.87865271	54.86233624	53.24415697	52.00488868
51.12432408	50.5813366	50.3539489	50.41940841	50.75427024
51.33448774	52.13551092	53.13239319	54.29990662	55.61266617
57.04526301	58.57240756	60.16908236	61.81070523	63.47330304



Task 3.4: Transormation in the data for Method 4

Method 4 is a little tricky and it is related to our transformation pattern. In this scenario we have x/100 as our pattern which after transforming we faced with numerical problems which are listed as below:

7.77703974e+05	6.47601556e+05	5.36509475e+05	4.42062623e+05
3.62137316e+05	2.94832261e+05	2.38450534e+05	1.91482592e+05
1.52590234e+05	1.20591517e+05	9.44466196e+04	7.32445558e+04
5.61907701e+04	4.25955874e+04	3.18634517e+04	2.34829391e+04
1.70174783e+04	1.20969019e+04	8.40953467e+03	5.69505713e+03
3.73793066e+03	2.36139014e+03	1.42211108e+03	8.05296631e+02
4.20299072e+02	1.96754395e+02	8.11195068e+01	3.36123047e+01
2.55831299e+01	3.72136230e+01	5.55299072e+01	7.27524414e+01
8.48995361e+01	9.06157227e+01	9.02960205e+01	8.53059082e+01
7.74561768e+01	6.85917969e+01	6.03374023e+01	5.39187012e+01



Output for method 4
it looks is similar to #1