Homework Assignment 3
Combined Work from
Diego Garcia-Olano, Vasko Lalkov and Yuege Xie

SDS 385 Statistical Models for Big Data

1. **Spectral Clustering**[1] In class we have seen k-means for estimating GMM's. Since your professor will not get a chance to actually cover her favorite clustering method aka spectral clustering, methinks this is a great opportunity to introduce you to it. There is a class of clustering algorithms, called spectral clustering algorithms, which has recently become quite popular. Many of these algorithms are quite easy to implement and perform well on certain clustering problems compared to more traditional methods like k-means. In this problem, we will try to develop some intuition about why these approaches make sense and implement one of these algorithms.

   Before beginning, we'll review a few basic linear algebra concepts you may find useful for some of the problems.

   - If A is a matrix, it has an v with eigenvalue λ if $Av = \lambda v$.
   - For any $m \times m$ symmetric matrix A, the *Singular Value Decomposition* of A yields a factorization of A into
   $$A = USU^T$$
   where U is an $m \times m$ orthogonal matrix (meaning that the columns are pairwise orthogonal). and $S = diag(|\lambda_1|, |\lambda_2|, \ldots, |\lambda_m|)$ where the λ_i are the eigenvalues of A.

   Given a set of m datapoints $x_1, \ldots, x_m$, the input to a spectral clustering algorithm typically consists of a matrix, A, of pairwise similarities between datapoints often called the *affinity matrix*. The choice of how to measure similarity between points is one which is often left to the practitioner. A very simple affinity matrix can be constructed as follows:

   $$A(i,j) = A(j,i) = \begin{cases} 1 & \text{if } d(x_i, x_j) < \Theta \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

   where $d(x_i, x_j)$ denotes the Euclidean distance between points x_i and x_j.

   The general idea of spectral clustering is to construct a mapping of the datapoints to an eigenspace of A with the hope that points are well separated in this eigenspace so that something simple like k-means applied to these new points will perform well.

   As an example, consider forming the affinity matrix for the dataset in Figure 1 using

[1]This problem was designed in collaboration with Jon Huang

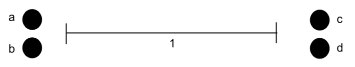


Figure 1: SimpSle dataset

Equation 1 with $\Theta = 1$. We have that

$$A = \left[\begin{array}{c|cccc} & a & b & c & d \\ \hline a & 1 & 1 & 0 & 0 \\ b & 1 & 1 & 0 & 0 \\ c & 0 & 0 & 1 & 1 \\ d & 0 & 0 & 1 & 1 \end{array}\right]$$

Now for this particular example, the clusters $\{a, b\}$ and $\{c, d\}$ show up as nonzero blocks in the affinity matrix. This is, of course, artificial, since we could have constructed the matrix A using any ordering of $\{a, b, c, d\}$. For example, another possibile affinity matrix for A could have been:

$$\tilde{A} = \left[\begin{array}{c|cccc} & a & c & b & d \\ \hline a & 1 & 0 & 1 & 0 \\ c & 0 & 1 & 0 & 1 \\ b & 1 & 0 & 1 & 0 \\ d & 0 & 1 & 0 & 1 \end{array}\right]$$

The key insight here is that the eigenvectors of matrices A and $\tilde{A}$ have the same entries (just permuted). The eigenvectors with nonzero eigenvalue of A are: $e_1 = (.7, .7, 0, 0)^T$, $e_2 = (0, 0, .7, .7)$. And the nonzero eigenvectors of $\tilde{A}$ are: $\tilde{e}_1 = (.7, 0, .7, 0)^T$, $\tilde{e}_2 = (0, .7, 0, .7)$. Spectral clustering embeds the original datapoints in a new space by using the coordinates of these eigenvectors. Specifically, it maps the point x_i to the point $(e_1(i), e_2(i), \ldots, e_k(i))$ where $e_1, \ldots, e_k$ are the top k eigenvectors of A. We refer to this mapping as the *spectral embedding.*

Algorithm description

Frequently, the affinity matrix is constructed as

$$A_{ij} = \begin{cases} 1 & \text{If } i, j \text{ are amongst } k \text{ nearest neighbors of each other} \\ 0 & \text{Otherwise} \end{cases} \tag{2}$$

The best that we can hope for in practice is a near block-diagonal affinity matrix. It can be shown in this case, that after projecting to the space spanned by the top k eigenvectors, points which belong to the same block are close to each other in a euclidean sense. We won't try to prove this, but using this intuition, you will implement one (of many) possible spectral clustering algorithms. This particular algorithm is described in

```
On Spectral Clustering: Analysis and an algorithm
Andrew Y. Ng, Michael I. Jordan, Yair Weiss (2001)
```

We won't try to justify every step, but see the paper if you are interested. The steps are as follows:

- Construct an affinity matrix A using Equation 2.
- Symmetrically 'normalize' the rows and columns of A to get a matrix N: such that $N(i,j) = \frac{A(i,j)}{\sqrt{d(i)d(j)}}$, where $d(i) = \sum_k A(i,k)$.
- Construct a matrix Y whose columns are the first k eigenvectors of N.
- Normalize each row of Y such that it is of unit length.
- Cluster the dataset by running k-means on the set of spectrally embedded points, where each row of Y is a datapoint.

(a) Generate a dataset with 10000 points, with 5000 coming from a Gaussian centered at $\mu_1(i) = 3/\sqrt{p}$ with $\Sigma_1 = I_p$ and the rest from a mean $\mu_2(i) = -3/\sqrt{p}$, for $i = 1, \ldots, p$, gaussian with $\Sigma_2 = I_p$. Create a k-nearest neighbor graph from this dataset and do Spectral clustering. For $p \in \{250, 500, 1000, 2000\}$, plot the time taken and the clustering accuracy of the following algorithms, averaged over 10 randomly generated datasets. You just have to write your own code for Spectral clustering, you can use available software like E2LSH or packages for kdtree for the rest. There are four algorithms, *Exact*, *JL+Exact*, *JL+KDtrees* and *E2LSH*. Use $k = 5$. You can also try out different k, its interesting to try this because too small or too large k can lead to a bad clustering accuracy. Remember that you can calculate the clustering accuracy because you generated the data and hence know the "latent" ground truth memberships.

   i. (Exact) Use the brute force k-nearest neighbor algorithm. If this is taking too long, you may omit the results, but please write explicitly how long it took, e.g. "my k-nn graph took over yyy hours to build when $p = xxx$ and so I gave up."

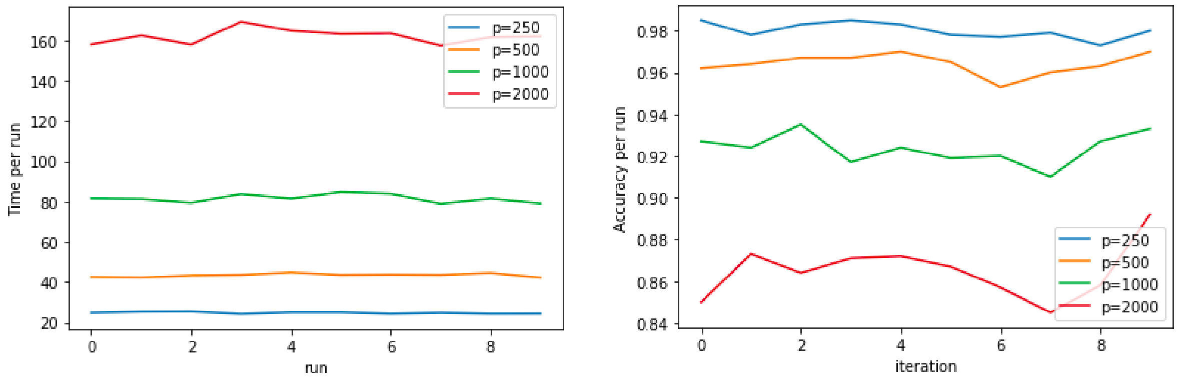
   ii. (JL+Exact, JL+KDtrees) Use the Johnson Lindenstrauss lemma to reduce the dimensionality of the data. Remember, for n points JL lets you project the datapoints into a $O(\log n/\epsilon^2)$ dimensional space with a multiplicative distortion of ϵ of the distances. Try out different ϵ values to generate the curves. For small ϵ, you would have higher accuracy and longer processing time, whereas for larger ϵ you would have lower accuracy and less computation time. Now use the KDtree algorithm and exact k-nearest neighbors to build the knn tree. There is a builtin function in Matlab using knnsearch which has an option of using the KDtree.

   iii. (E2LSH) Use E2LSH to implement Locality sensitive hashing to obtain the nearest neighbor graph. You can find a matlab package here: `http://ttic.uchicago.edu/~gregory/download.html`.

ANSWER: For the experiments, I ran each configuration for 10 runs but with 1000 data points (divided into two sets of 500 points coming from each of the Gaussian mixtures) to speed things up as time was becoming an issue on my machine. This was only an issue for the part 1 EXACT section, as the other projection based methods go much faster in general, but in order to keep the comparison even I kept the data points size the same. Additionally for the final step, as opposed to running kmeans with k=2 to cluster and get predictions, I ran knn again and used a majority vote of neighbors to classify each point. This was an accident on my end, but this method is effectively the same given the reduced space, and on looking at how kmeans clustering on Y compared over a few different runs I noticed they were very similar.

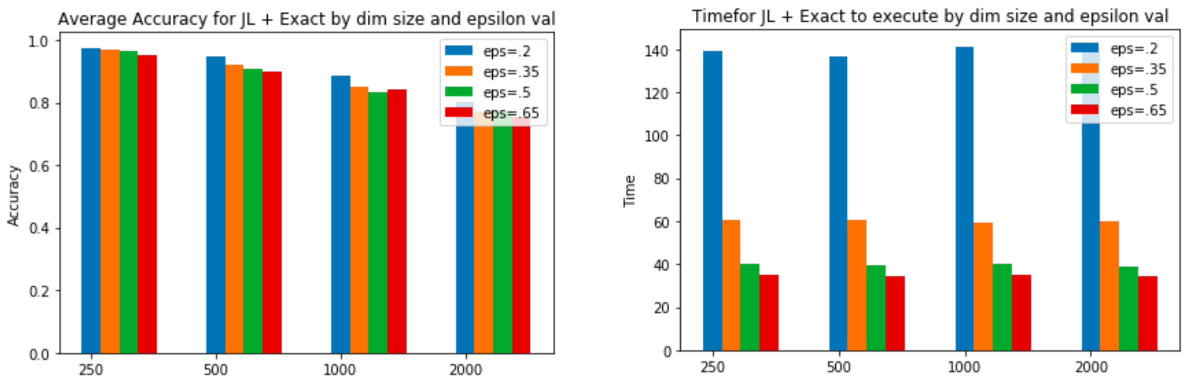
For the Exact spectral clustering case, we see that running it on higher dimensional data leads to longer computation time and less accuracy though it still does relatively well given how high of a dimension the data is and how its generated. For lower dimensional space, the method is both faster and more accurate.

Figure 2: Time & Accuracy plots for 10 runs of Exact Spectral Clustering with k=5



Looking at Figure 3, we see using the Johnson Lindenstrauss lemma to first reduce our data using different values of epsilon and then performing exact spectral clustering leads to comparable results in terms of accuracy for lower dimensional data depending on the choice of epsilon (ie, lower values of epsilon take longer to run, but lead to better performance than higher epsilon). The time benefit is evident when working on data in higher dimensions, and its possible that using an even smaller value of epsilon would lead to comparable accuracy, but at that point the time improvement would be totally lost

Figure 3: Time & Accuracy plots for 10 runs of JL + Exact Spectral Clustering



In Figure 4, we see that for JL + KDTrees, we see a vast improvement in terms of processing speed compared with both prior methods, but at a cost of accuracy across the board though this is particularly pronounced in higher dimensions. It possible that using even lower values of epsilon would help, but in looks at epsilon=.15 the performance was only slightly improved. Figure 5 shows a side by side comparison of the first 3 methods. Exact Spectral Clustering outperforms across the board and is still able to handle higher dimensions in reasonable time (though i imagine this time would be exaggerated had i used 10k points at which point the tradeoff of accuracy/time using either JL + Exact and JL + KD trees would be something to consider.)

Figure 4: Time & Accuracy plots for 10 runs of JL + KDTrees

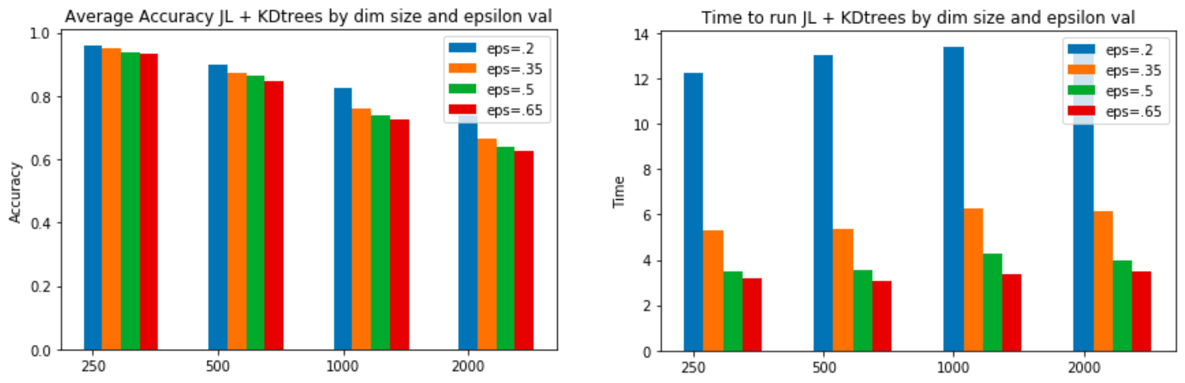
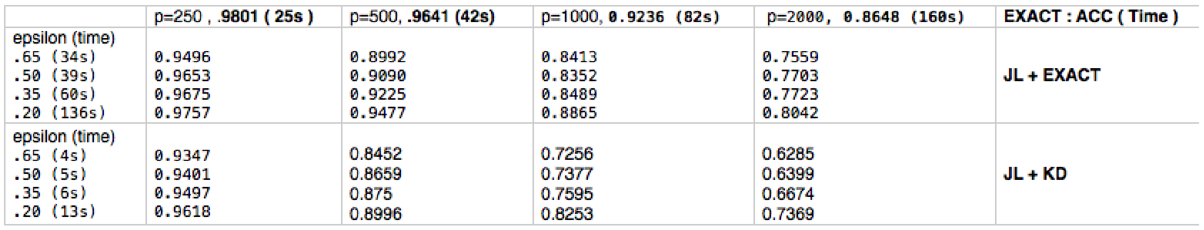


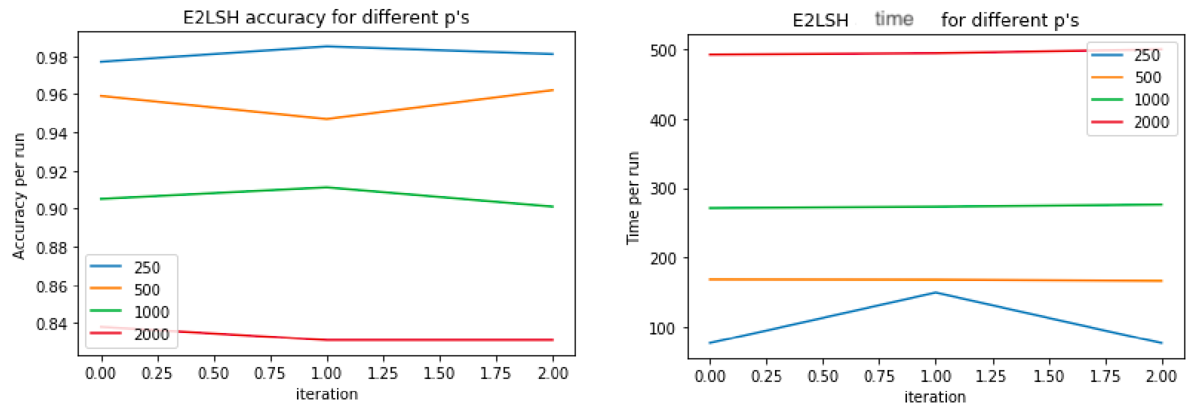
Figure 5: Table for Exact, JL + Exact Spectral Clustering and JL + KDtrees



	p=250 , **.9801 (25s)**	p=500, **.9641 (42s)**	p=1000, **0.9236 (82s)**	p=2000, **0.8648 (160s)**	**EXACT : ACC (Time)**
epsilon (time)					
.65 (34s)	0.9496	0.8992	0.8413	0.7559	
.50 (39s)	0.9653	0.9090	0.8352	0.7703	**JL + EXACT**
.35 (60s)	0.9675	0.9225	0.8489	0.7723	
.20 (136s)	0.9757	0.9477	0.8865	0.8042	
epsilon (time)					
.65 (4s)	0.9347	0.8452	0.7256	0.6285	
.50 (5s)	0.9401	0.8659	0.7377	0.6399	**JL + KD**
.35 (6s)	0.9497	0.875	0.7595	0.6674	
.20 (13s)	0.9618	0.8996	0.8253	0.7369	

For the final section on using E2LSH to implement Locality sensitive hashing to obtain the nearest neighbor graph, we used the FALCONN library which is from a 2015 NIPS paper "FAst Lookups of Cosine and Other Nearest Neighbors" that is based on E2LSH and has a Python wrapper (i wrote to Dr. Sarkar about this the weekend before the due date and she never got back to me so I just went with it since its the most comparable thing I could find in Python and I don't have matlab on my laptop). In Figure 6, we can see the results of using E2LSH for our problem in both accuracy and time.

Figure 6: E2LSH



The time graph is slightly misleading because in order to obtain its high accuracy, the algorithm first does a few iterations of hyper parameter tuning to determine what is the ideal number of probes to use in order to get high accuracy. Figure 7 shows how this tuning affects the time and accuracy of three runs when p=250 (less probes tested equals less time).

Figure 7: E2LSH probes output

```
1000 -> 0.868
2000 -> 0.953
4000 -> 0.971
8000 -> 0.976
16000 -> 0.977
{1000: 0.868, 2000: 0.953, 4000: 0.971, 8000: 0.976, 16000: 0.977}
select num probes 32000 with accuracy 0.977
----Inner elapsed time for p 250  and run i 0 is  77.13687205314636 with acc 0.977
1000 -> 0.873
2000 -> 0.952
4000 -> 0.964
8000 -> 0.968
16000 -> 0.983
32000 -> 0.985
64000 -> 0.981
{1000: 0.873, 2000: 0.952, 4000: 0.964, 8000: 0.968, 16000: 0.983, 32000: 0.985, 64000: 0.981}
select num probes 32000 with accuracy 0.985
----Inner elapsed time for p 250  and run i 1 is  149.5004642009735 with acc 0.985
1000 -> 0.873
2000 -> 0.948
4000 -> 0.97
8000 -> 0.981
16000 -> 0.978
{1000: 0.873, 2000: 0.948, 4000: 0.97, 8000: 0.981, 16000: 0.978}
select num probes 8000 with accuracy 0.981
----Inner elapsed time for p 250  and run i 2 is  77.09041357040405 with acc 0.981
--Elapsed time for p 250 is  303.7324945926666
```

2. We are going to learn and implement the power method in this problem. Let the eigenvalues of a square symmetric matrix $A \in \mathbb{R}^{n\times n}$ be given by $|\lambda_1| > |\lambda_2| > \dots$. We will assume that the eigenvalues are all different for convenience of analysis. Start with some

vector $\boldsymbol{q}^{(0)}$. Now compute the following:

$$\boldsymbol{z}^{(k)} = A\boldsymbol{q}^{(k-1)} \tag{3}$$

$$\boldsymbol{q}^{(k)} = \frac{\boldsymbol{z}^{(k)}}{\|\boldsymbol{z}^{(k)}\|} \tag{4}$$

$$\nu^{(k)} = (\boldsymbol{q}^{(k)})^T A\boldsymbol{q}^{(k)} \tag{5}$$

(a) Prove that for $k > 1$,

$$\boldsymbol{q}^{(k)} = \frac{A^k\boldsymbol{q}^{(0)}}{\|A^k\boldsymbol{q}^{(0)}\|}.$$

You can use induction to do this.

At the start we know the vector $\boldsymbol{q}^{(0)}$ and we can write the following for $k = 1$:

$$\boldsymbol{q}^{(1)} = \frac{A\boldsymbol{q}^{(0)}}{\|A\boldsymbol{q}^{(0)}\|}.$$

Let us now take $k = 2$.

$$\boldsymbol{q}^{(2)} = \frac{A\boldsymbol{q}^{(1)}}{\|A\boldsymbol{q}^{(1)}\|} = \frac{A\frac{A\boldsymbol{q}^{(0)}}{\|A\boldsymbol{q}^{(0)}\|}}{\|A\frac{A\boldsymbol{q}^{(0)}}{\|A\boldsymbol{q}^{(0)}\|}\|} = \frac{\frac{A^2\boldsymbol{q}^{(0)}}{\|A\boldsymbol{q}^{(0)}\|}}{\frac{\|A^2\boldsymbol{q}^{(0)}\|}{\|A\boldsymbol{q}^{(0)}\|}} = \frac{A^2\boldsymbol{q}^{(0)}}{\|A^2\boldsymbol{q}^{(0)}\|}$$

Thus, the conjecture is true for $k = 2$. Now we assume that it is true for some k. Then, we have the following.

$$\boldsymbol{q}^{(k)} = \frac{A^k\boldsymbol{q}^{(0)}}{\|A^k\boldsymbol{q}^{(0)}\|}$$

Finally we investigate the case with $k + 1$.

$$\boldsymbol{q}^{(k+1)} = \frac{A\boldsymbol{q}^{(k)}}{\|A\boldsymbol{q}^{(k)}\|} = \frac{A\frac{A^k\boldsymbol{q}^{(0)}}{\|A^k\boldsymbol{q}^{(0)}\|}}{\|A\frac{A^k\boldsymbol{q}^{(0)}}{\|A^k\boldsymbol{q}^{(0)}\|}\|} = \frac{\frac{A^{k+1}\boldsymbol{q}^{(0)}}{\|A^k\boldsymbol{q}^{(0)}\|}}{\frac{\|A^{k+1}\boldsymbol{q}^{(0)}\|}{\|A^k\boldsymbol{q}^{(0)}\|}} = \frac{A^{k+1}\boldsymbol{q}^{(0)}}{\|A^{k+1}\boldsymbol{q}^{(0)}\|}$$

Starting with the vector $\boldsymbol{q}^{(0)}$, the conjecture is true for $k = 1$ and $k = 2$. Assuming the for the case k the conjecture is true, we obtain that it is also true for the next step, $k+1$. Thus, by induction, we have proven that $\boldsymbol{q}^{(k)} = \frac{A^k\boldsymbol{q}^{(0)}}{\|A^k\boldsymbol{q}^{(0)}\|}$.

(b) Now let the eigenvectors of A be $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$ and let

$$\boldsymbol{q}^{(0)} = \sum_{i=1}^{n} \alpha_i\boldsymbol{x}_i, \quad \boldsymbol{y}^{(k)} = \sum_{i=2}^{n} \frac{\alpha_i}{\alpha_1}\left(\frac{\lambda_i}{\lambda_1}\right)^k \boldsymbol{x}_i.$$

Show that

$$\|\boldsymbol{q}^{(k)} - \langle\boldsymbol{q}^{(k)}, \boldsymbol{x}_1\rangle\boldsymbol{x}_1\| \leq C\left|\frac{\lambda_2}{\lambda_1}\right|^k$$

Let us begin writing the terms inside the norm on the left.

$$\boldsymbol{q}^{(k)} = \frac{A^k \boldsymbol{q}^{(0)}}{\|A^k \boldsymbol{q}^{(0)}\|} = \frac{A^k \sum_{i=1}^n \alpha_i x_i}{\|A^k \boldsymbol{q}^{(0)}\|} = \frac{\sum_{i=1}^n \alpha_i A^k x_i}{\|A^k \boldsymbol{q}^{(0)}\|} = \frac{\sum_{i=1}^n \alpha_i \lambda_i^k x_i}{\|A^k \boldsymbol{q}^{(0)}\|}$$

$$\langle \boldsymbol{q}^{(k)}, \boldsymbol{x}_1 \rangle \boldsymbol{x}_1 = \frac{1}{\|A^k \boldsymbol{q}^{(0)}\|} \sum_{i=1}^n \alpha_i \lambda_i^k x_i^\top x_1 x_1$$

Without loss of generality, we assume that $\|x_i\| = 1$. We also know that $\langle x_i, x_j \rangle = 0$, $\forall i \neq j$. Thus we get the following.

$$\langle \boldsymbol{q}^{(k)}, \boldsymbol{x}_1 \rangle \boldsymbol{x}_1 = \frac{1}{\|A^k \boldsymbol{q}^{(0)}\|} \alpha_1 \lambda_1^k x_1$$

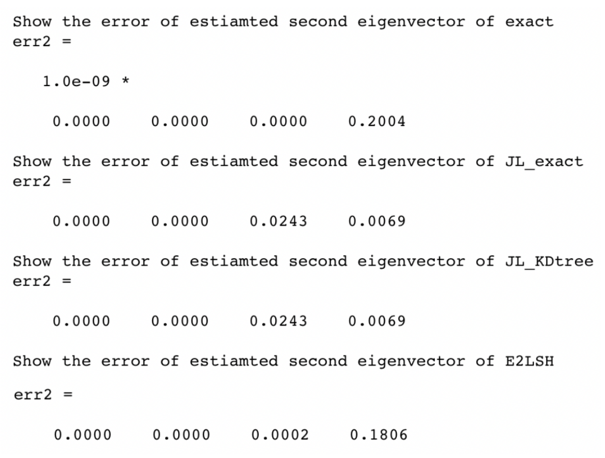
Now we put those two expressions together.

$$\begin{aligned}
\|\boldsymbol{q}^{(k)} - \langle \boldsymbol{q}^{(k)}, \boldsymbol{x}_1 \rangle \boldsymbol{x}_1\| &= \left\| \frac{1}{\|A^k \boldsymbol{q}^{(0)}\|} \sum_{i=1}^n \alpha_i \lambda_i^k x_i - \frac{1}{\|A^k \boldsymbol{q}^{(0)}\|} \alpha_1 \lambda_1^k x_1 \right\| \\
&= \left\| \frac{1}{\|A^k \boldsymbol{q}^{(0)}\|} \sum_{i=2}^n \alpha_i \lambda_i^k x_i \right\| \\
&= \frac{1}{\|A^k \boldsymbol{q}^{(0)}\|} \left\| \sum_{i=2}^n \alpha_i \lambda_i^k x_i \right\| \\
&\leq \frac{1}{\|A^k \boldsymbol{q}^{(0)}\|} \sum_{i=2}^n \alpha_i \lambda_i^k \|x_i\| \\
&= \frac{1}{\|A^k \boldsymbol{q}^{(0)}\|} \sum_{i=2}^n \alpha_i \lambda_i^k \\
&\leq \frac{1}{\|A^k \boldsymbol{q}^{(0)}\|} \lambda_2^k \sum_{i=2}^n \alpha_i \\
&= \frac{|\lambda_1^k| |\sum_{i=2}^n \alpha_i|}{\|A^k \boldsymbol{q}^{(0)}\|} \left| \frac{\lambda_2}{\lambda_1} \right|^k
\end{aligned}$$

Since the term in the front is a constant, we have shown the inequality provided in the question.

3. Now, write your own routine for power iteration and apply it to compute the second eigenvector of the nearest neighbor graphs you have built in the last question. For each of them, show the error (you should adjust for signs since an eigenvector can be multiplied by 1 or -1 but it still remains an eigenvector) of your estimated vector and the second eigenvector computed using matlab or R.

   Solution. See "p3_power-iteration.html" for codes and results. The printed numerical errors and corresponding plots are in Figure 8. The errors are small and acceptable.



```
Show the error of estiamted second eigenvector of exact
err2 =

   1.0e-09 *

    0.0000    0.0000    0.0000    0.2004

Show the error of estiamted second eigenvector of JL_exact
err2 =

    0.0000    0.0000    0.0243    0.0069

Show the error of estiamted second eigenvector of JL_KDtree
err2 =

    0.0000    0.0000    0.0243    0.0069

Show the error of estiamted second eigenvector of E2LSH

err2 =

    0.0000    0.0000    0.0002    0.1806
```

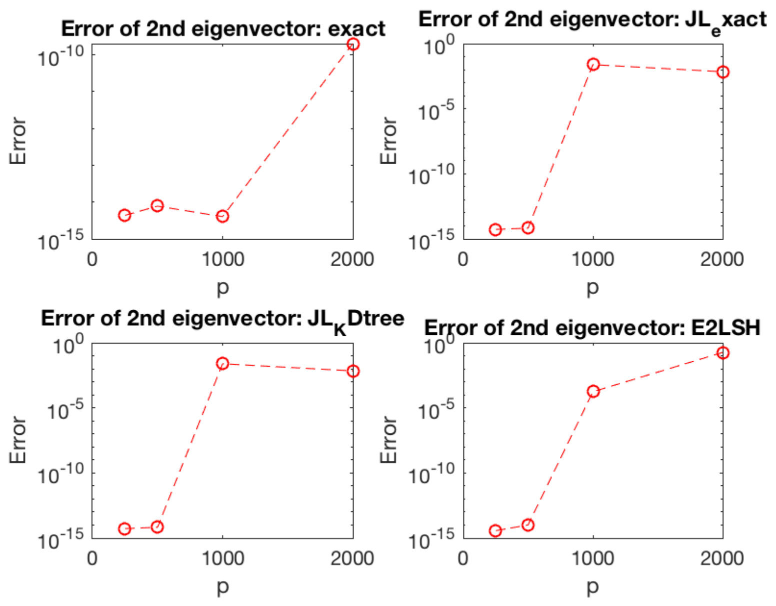


Figure 8: Plot for 3. Error of estimated second eigenvector compared to the results calculated using matlab package. ■

4. Last, but not the least, we will learn kernel PCA to deal with non-linear decision boundaries. Recall PCA? There you first centered your data to get $\tilde{X}$, computed covariance matrix, and then compute top K eigenvectors V_k of this and project a datapoint $\tilde{x}$ to get $\tilde{x}^T V_k$. Now we will directly compute these projections .

   (a) Assume that you are mapping the datapoints to a different feature space $\phi(\boldsymbol{x}) \in \mathbb{R}^N$ and $\sum_i \phi(\boldsymbol{x}_i) = 0$. While we will not do this explicitly, let us follow through the steps of PCA applied on this new feature space. Create a new matrix $\phi(X) \in \mathbb{R}^{n \times N}$. Let $\boldsymbol{v}$ be the first eigenvector of the covariance matrix in this feature space, i.e.

$$\sum_{i=1}^{n} \phi(\boldsymbol{x}_i)\phi(\boldsymbol{x}_i)^T \boldsymbol{v} = \lambda \boldsymbol{v}. \tag{6}$$

Show that $\boldsymbol{v}$ can be written as

$$\boldsymbol{v} = \phi(X)^T \boldsymbol{a} \tag{7}$$

So finding $\boldsymbol{v}$ boils down to finding $\boldsymbol{a}$.

Proof. Let $a_i = \frac{\phi(\boldsymbol{x}_i)^T \boldsymbol{v}}{\lambda}, \forall i \in [n]$ and $\boldsymbol{a} = [a_i, \dots, a_n]^T$, then from Eq (6),

$$\boldsymbol{v} = \frac{1}{\lambda} \sum_{i=1}^{n} \phi(\boldsymbol{x}_i)\phi(\boldsymbol{x}_i)^T \boldsymbol{v} = \sum_{i=1}^{n} a_i \phi(\boldsymbol{x}_i) = \phi(X)^T \boldsymbol{a}$$

since $\phi(X)^T = [\phi(\boldsymbol{x}_1) \ \ \phi(\boldsymbol{x}_2) \ \ \cdots \ \ \phi(\boldsymbol{x}_n)] \in \mathbb{R}^{N \times n}$. □

(b) Now show that if you could get $\boldsymbol{a}$, the projection of the data (in the new space) on this direction is given by:

$$\Phi(X)\boldsymbol{v} = K\boldsymbol{a},$$

where $K(i,j) = \phi(\boldsymbol{x}_i)^T \phi(\boldsymbol{x}_j)$. Now we will get $\boldsymbol{a}$.

Proof. Since from (a), $\boldsymbol{v} = \phi(X)^T \boldsymbol{a}$, then

$$\phi(X)\boldsymbol{v} = \phi(X)\phi(X)^T \boldsymbol{a} = K\boldsymbol{a}$$

where $K = \phi(X)\phi(X)^T$. Since

$$\phi(X) = \begin{bmatrix} — & \phi(\boldsymbol{x}_1)^T & — \\ — & \phi(\boldsymbol{x}_2)^T & — \\ & \vdots & \\ — & \phi(\boldsymbol{x}_n)^T & — \end{bmatrix}_{n \times N} \quad \phi(X)^T = \begin{bmatrix} | & | & & | \\ \phi(\boldsymbol{x}_1) & \phi(\boldsymbol{x}_2) & \dots & \phi(\boldsymbol{x}_n) \\ | & | & & | \end{bmatrix}_{N \times n}$$

we have $K(i,j) = \phi(\boldsymbol{x}_i)^T \phi(\boldsymbol{x}_j)$. □

(c) Now plug in Eq (7) to Eq (6), and left multiply by $\phi(\boldsymbol{x}_k)^T$ to get:

$$K\boldsymbol{a} = \lambda \boldsymbol{a}.$$

You can assume that K is invertible.

Proof. plug in Eq (7) to Eq (6), since $\sum_{i=1}^{n} \phi(\boldsymbol{x}_i)\phi(\boldsymbol{x}_i)^T = \phi(X)^T \phi(X)$, we have

$$\sum_{i=1}^{n} \phi(\boldsymbol{x}_i)\phi(\boldsymbol{x}_i)^T \boldsymbol{v} = (\sum_{i=1}^{n} \phi(\boldsymbol{x}_i)\phi(\boldsymbol{x}_i)^T)\boldsymbol{v} = \phi(X)^T \phi(X)\phi(X)^T \boldsymbol{a} = \lambda \phi(X)^T \boldsymbol{a}$$

Then, left multiply by $\phi(X)$, since $K = \phi(X)\phi(X)^T$ and K is invertible, we have

$$\phi(X)\phi(X)^T \phi(X)\phi(X)^T \boldsymbol{a} = \lambda \phi(X)\phi(X)^T \boldsymbol{a}$$

$$\iff K^2 \boldsymbol{a} = \lambda K \boldsymbol{a} \iff K\boldsymbol{a} = \lambda \boldsymbol{a}$$

□

(d) But typically we don't have centered features. So instead of $\phi(\boldsymbol{x}_i)$ we should work with $\psi(\boldsymbol{x}_i) = \phi(\boldsymbol{x}_i) - \frac{\sum_j \phi(\boldsymbol{x}_j)}{n}$. Show that the kernel matrix built from these centered feature vectors equal $\tilde{K} = K - K\mathbf{1}\mathbf{1}^T/n - \mathbf{1}\mathbf{1}^T/nK + \mathbf{1}^TK\mathbf{1}/n^2\mathbf{1}\mathbf{1}^T$

Proof. Rewrite $\psi(\boldsymbol{x}_i) = \phi(\boldsymbol{x}_i) - \frac{\sum_j \phi(\boldsymbol{x}_j)}{n} = \phi(\boldsymbol{x}_i) - \frac{1}{n}\phi(X)^T\mathbf{1}$, by the form of $\phi(X)^T$ in (b),

$$\psi(X)^T = \begin{bmatrix} | & | & & | \\ \psi(\boldsymbol{x}_1) & \psi(\boldsymbol{x}_2) & \dots & \psi(\boldsymbol{x}_n) \\ | & | & & | \end{bmatrix} = \begin{bmatrix} \phi(\boldsymbol{x}_1) - \frac{1}{n}\phi(X)^T\mathbf{1} & \dots & \phi(\boldsymbol{x}_n) - \frac{1}{n}\phi(X)^T\mathbf{1} \end{bmatrix}$$
$$= \phi(X)^T - \frac{1}{n}\phi(X)^T\mathbf{1}\mathbf{1}^T$$

Then, plug in $\psi(X)$ into the definition of $\tilde{K}$, we have

$$\begin{aligned}
\tilde{K} &= \psi(X)\psi(X)^T \\
&= [\phi(X)^T - \frac{1}{n}\phi(X)^T\mathbf{1}\mathbf{1}^T]^T[\phi(X)^T - \frac{1}{n}\phi(X)^T\mathbf{1}\mathbf{1}^T] \\
&= [\phi(X) - \frac{1}{n}\mathbf{1}\mathbf{1}^T\phi(X)][\phi(X)^T - \frac{1}{n}\phi(X)^T\mathbf{1}\mathbf{1}^T] \\
&= \phi(X)\phi(X)^T - \frac{1}{n}\phi(X)\phi(X)^T\mathbf{1}\mathbf{1}^T - \frac{1}{n}\mathbf{1}\mathbf{1}^T\phi(X)\phi(X)^T + \frac{1}{n^2}\mathbf{1}\mathbf{1}^T\phi(X)\phi(X)^T\mathbf{1}\mathbf{1}^T \\
&= K - \frac{1}{n}K\mathbf{1}\mathbf{1}^T - \frac{1}{n}\mathbf{1}\mathbf{1}^TK + \frac{1}{n^2}\mathbf{1}(\mathbf{1}^TK\mathbf{1})\mathbf{1}^T
\end{aligned}$$

Since $\mathbf{1}^TK\mathbf{1}$ is a scalar, we have $\tilde{K} = K - \frac{1}{n}K\mathbf{1}\mathbf{1}^T - \frac{1}{n}\mathbf{1}\mathbf{1}^TK + \frac{(\mathbf{1}^TK\mathbf{1})}{n^2}\mathbf{1}\mathbf{1}^T$. □

(e) So the final algorithm is to build $\tilde{K}$ matrix from the data using your choice of a kernel, and then compute top eigenvector of this matrix and project K on that direction. Download the two parabola dataset. Plot the first principal component using PCA.

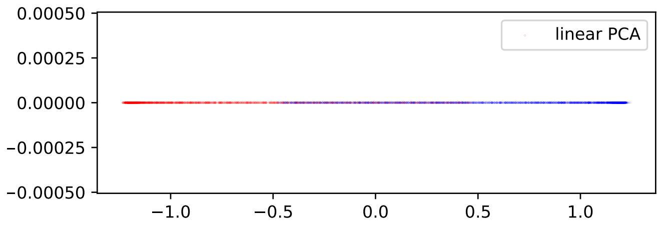
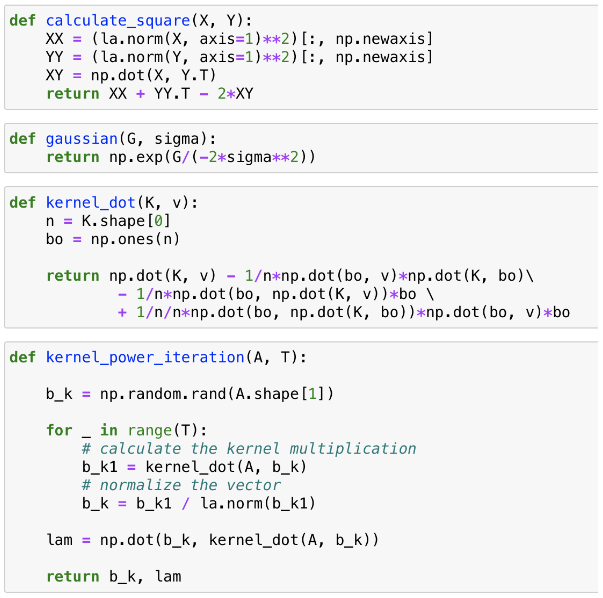


Figure 9: Plot for (e). First eigenvector using linear PCA

Solution. As Figure 9 shows, the two clusters are not separated well. ■

(f) Now do Kernel PCA with the RBF kernel $K(i,j) = \exp(-\|\boldsymbol{x}_i - \boldsymbol{x}_j\|^2/2\sigma^2)$. Change your power method algorithm so that you do not need to compute the $\tilde{K}$ matrix, but only the K matrix. Provide the Pseudocode.



```python
def calculate_square(X, Y):
    XX = (la.norm(X, axis=1)**2)[:, np.newaxis]
    YY = (la.norm(Y, axis=1)**2)[:, np.newaxis]
    XY = np.dot(X, Y.T)
    return XX + YY.T - 2*XY
```

```python
def gaussian(G, sigma):
    return np.exp(G/(-2*sigma**2))
```

```python
def kernel_dot(K, v):
    n = K.shape[0]
    bo = np.ones(n)

    return np.dot(K, v) - 1/n*np.dot(bo, v)*np.dot(K, bo)\
            - 1/n*np.dot(bo, np.dot(K, v))*bo \
            + 1/n/n*np.dot(bo, np.dot(K, bo))*np.dot(bo, v)*bo
```

```python
def kernel_power_iteration(A, T):

    b_k = np.random.rand(A.shape[1])

    for _ in range(T):
        # calculate the kernel multiplication
        b_k1 = kernel_dot(A, b_k)
        # normalize the vector
        b_k = b_k1 / la.norm(b_k1)

    lam = np.dot(b_k, kernel_dot(A, b_k))

    return b_k, lam
```

Figure 10: Plot for (f). Code for Kernel PCA with power method.

Solution. The code is in Figure 10, and ”pseudo code” description is

- calculate pairwise distance using matrix form (1st block)
- generate not centered Gaussian kernel matrix (2nd block)
- use ”kernel dot” to do power iteration, using only matrix vector product with K and $\mathbf{1}$ (3rd block), then the rest routine is power method (4th block).

■

(g) Try different values of σ, and report the one which returns a first kernel PC such that the two classes are separated along this direction.

Solution. See ”p4_kernel-pca.html” for codes and results. Figure 11 shows the results using kernel PCA with different σ. As it shows, when σ is between 0.1 and 0.2, the two classes are separated pretty well.

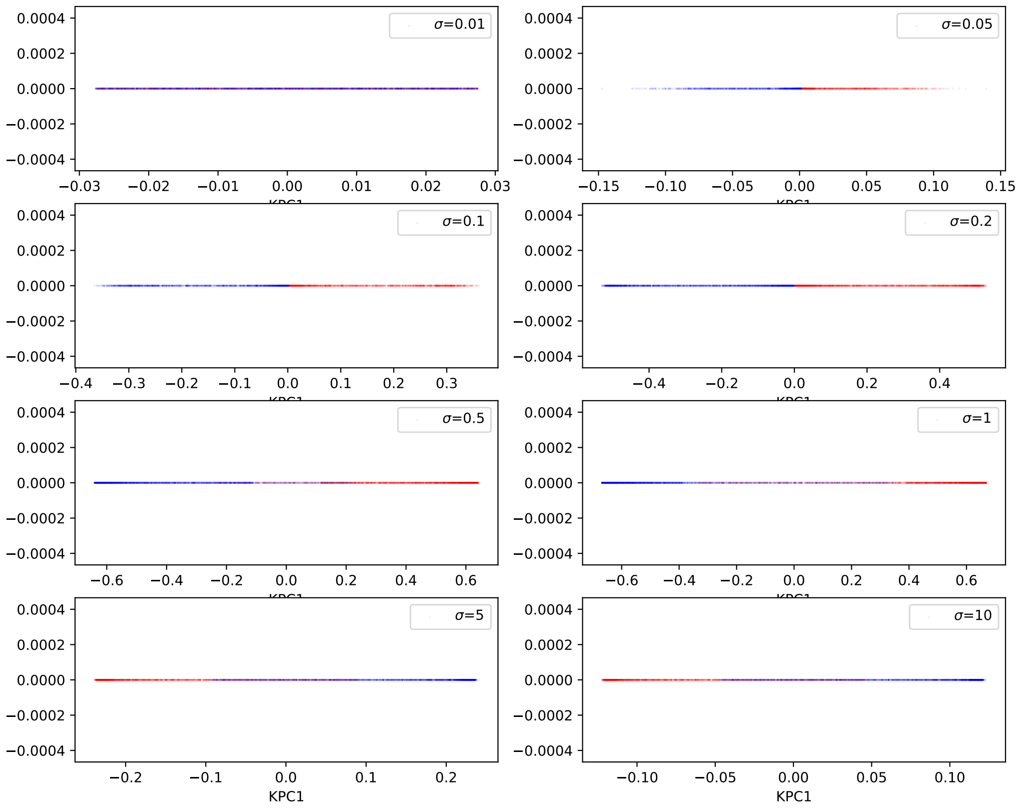


Figure 11: Plot for (g). First eigenvector using kernel PCA with gaussian kernel with differentσ values: $\sigma = 0.01, 0.05, 0.1, 0.2, 0.5, 1, 5, 10$.

■

(h) What do you think is the relationship of Spectral Clustering with Kernel PCA? Give your answer in 4-5 lines.

In spectral clustering, we are essentially mapping the data points to the eigenspace of the similarity matrix A, that is based on the similarities between the data points. In this new space, the points are hopefully well separated.

Kernel PCA is actually very similar to this, but instead of projecting the data points to the eigenspace of a similarity matrix, now we project it to the kernel space, defined by the kernel function that we use.

Thus, the similarity is that through projecting the original data points in a new space, we hope to get a good separation between the points. The difference is that in Kernal PCA, we have the parameter σ that we have to adjust for. This perhaps provides more freedom in constructing how well separated we want the points to be, but it also might pose some difficulties.