Final Project

Software Project (0368-2161)

Due Date: 24/04/2022, NO EXTENSIONS!

1 Introduction

In this project you will implement a version of the normalized spectral clustering algorithm. This document starts by introducing the mathematical basis and algorithms for the project, and then describes the code and implementation requirements. We are not going to provide a tester for the project – implementation and correctness are up to you. You will be graded for code modularity, design, readability, and performance.

Normalized Spectral Clustering We present the Normalized Spectral Clustering algorithm based on [1, 2]. Given a set of n points $X = x_1, x_2, \ldots, x_N \in R^d$ the algorithm is:

Algorithm 1 The Normalized Spectral Clustering Algorithm

1: Form the weighted adjacency matrix W from X
2: Compute the normalized graph Laplacian L_{norm}
3: Determine k and obtain the first k eigenvectors $u_1, \ldots, u_k$ of L_{norm}
4: Let $U \in R^{n \times k}$ be the matrix containing the vectors $u_1, \ldots, u_k$ as columns
5: Form the matrix $T \in R^{n \times k}$ from U by renormalizing each of U's rows to have unit length, that is set $t_{ij} = u_{ij} / (\sum_j u_{ij}^2)^{1/2}$
6: Treating each row of T as a point in R^k, cluster them into k clusters via the K-means algorithm

A few things to keep in mind:

- Step 3 - eigenvalues must be ordered increasingly, respecting multiplicities (for example, all the first five eigenvalues could be equal to 0). By "the first k eigenvectors" we refer to the eigenvectors that correspond to the k smallest eigenvalues. Determining k would be based on the eigengap heuristic described at subsection 1.3 or given as an input.
- Step 6:
  - (Python): you should use the K-means implementation from the homework assignments, with the needed adjustments to fit the project. You are expected to use the K-means++ initialization when applying K-means as in HW_2. The `MAX_ITER` variable should be set to 300.

1.1 Graph Representation

We aim at creating an undirected graph $G = (V, E; W)$, that will represent the n points at hand. Each point x_i is viewed as a vertex v_i (also known as node) to produce $V = \{v_1, v_2, \ldots, v_n\}$. A common mapping is to set $v_i = i$. The set of edges (also known as arcs) E will be the union of all connected pairs $\{v_i, v_j\}$. Next, we present the way to decide the weights of the graph and the structure of the graph.

1.1.1 The Weighted Adjacency Matrix

Let w_{ij} represent the weight of the connection between v_i and v_j. Only if $w_{ij} > 0$, we define an edge between v_i and v_j. The weighted adjacency matrix $W \in R^{n \times n}$ (also referred to as the affinity matrix) is defined as $W = (w_{ij})_{i,j=1,\ldots,n}$. The weights are symmetric ($w_{ij} = w_{ji}$) and non-negative ($w_{ij} \geq 0$). As we do not allow self loops, we set $w_{ii} = 0$ for all i's. In order to create a fully connected graph, the rest of the values are set to:

$$w_{ij} = \exp\left(-\frac{\|x_i - x_j\|_2}{2}\right)$$

We denote by $\|\|_2$ the ℓ_2-norm or the Euclidean norm ($\|a - b\|_2 = \sqrt{\sum_{i=1}^{d}(a_i - b_i)^2}$).

1.1.2 The Diagonal Degree Matrix

The diagonal degree matrix $D \in R^{n \times n}$ is defined as $D = (d_{ij})_{i,j=1,\ldots,n}$, such that:

$$d_{ij} = \begin{cases} \sum_{z=1}^{n} w_{iz}, & \text{if } i = j \\ 0, & \text{otherwise} \end{cases}$$

We get that i-th element along the diagonal equals to the sum of the i-th row of W. In essence, D's diagonal elements represent the sum of weights that lead to vertex v_i. Note that

$$D^{-\frac{1}{2}} = \begin{pmatrix} \frac{1}{\sqrt{d_{11}}} & 0 & \ldots & 0 \\ 0 & \frac{1}{\sqrt{d_{22}}} & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & \frac{1}{\sqrt{d_{nn}}} \end{pmatrix}$$

1.1.3 The Normalized Graph Laplacian

The normalized graph Laplacian $L_{norm} \in R^{n \times n}$ is defined as

$$L_{norm} = I - D^{-\frac{1}{2}} W D^{-\frac{1}{2}}$$

The reason we are interested in L_{norm} is that it has all eigenvalues $\lambda_1 \leq \ldots \leq \lambda_n$ are real and non-negative.

1.2 Finding Eigenvalues and Eigenvectors

In this section, we lay the foundation needed to find **all** of the eigenvectors and eigenvalues of a real, symmetric, full rank matrix. Then we present a heuristic that utilizes the eigenvalues to determine

the number of k clusters the data holds. It is a mathematical approach very similar to the visual "elbow" method presented in the second homework assignment.

1.2.1 Jacobi algorithm

The Jacobi eigenvalue algorithm is an iterative method for the calculation of the eigenvalues and eigenvectors of a real symmetric matrix (a process known as diagonalization).

1. **Procedure**
   (a) Build a rotation matrix P (as explained below).
   (b) Transform the matrix A to:
   $$A' = P^T A P$$
   $$A = A'$$
   (c) Repeat a,b until A' is diagonal matrix.
   (d) The diagonal of the final A' is the eigenvalues of the original A.
   (e) Calculate the eigenvectors of A by multiplying all the rotation matrices:
   $$V = P_1 P_2 P_3 \dots$$

2. **Rotation Matrix** P
   Let S be a symmetric matrix and P be the Jacobi rotation matrix of the form:
   $$P = \begin{pmatrix} 1 & & & & & & \\ & \dots & & & & & \\ & & c & \dots & s & & \\ & & \vdots & 1 & \vdots & & \\ & & -s & \dots & c & & \\ & & & & & \dots & \\ & & & & & & 1 \end{pmatrix}$$
   Here all the diagonal elements are unity except for the two elements c in rows (and columns) i and j and all off-diagonal elements are zero except the two elements s and $-s$. Also, $s = \sin\phi$ and $c = \cos\phi$.

3. **Pivot**
   The A_{ij} is the off-diagonal element with **the largest absolute value**.

4. **Obtain** c, t
   $$\theta = \cot 2\phi = \frac{A_{jj} - A_{ii}}{2A_{ij}}$$
   $$t = \frac{sign(\theta)}{|\theta| + \sqrt{\theta^2 + 1}}$$
   $$c = \frac{1}{\sqrt{t^2 + 1}}, \quad s = tc$$
   Note: We define $sign(0) = 1$

5. **Convergence**: Let $off(A)^2$ and $off(A')^2$ be the sum of squares of all off-diagonal elements of A and A' respectively. Then the square of off diagonal elements of A is

$$off(A)^2 = \sum_{i=1}^{N} \sum_{j=1, j\neq i}^{N} a_{ij}^2$$

At step c in the above procedure, we define convergence as follow:

$$off(A)^2 - off(A')^2 \leq \epsilon$$

We will be using $\epsilon = 1.0 \times 10^{-5}$ OR maximum number of rotations = 100

6. **Relation between A and A'**:
After each transformation of step 2, the modified elements of A are only the i and j rows and columns. Therefore, from the symmetry of A, we obtain the following formula to calculate A':

$$\begin{aligned}
a'_{ri} &= ca_{ri} - sa_{rj} \qquad r \neq i, j \\
a'_{rj} &= ca_{rj} + sa_{ri} \qquad r \neq i, j \\
a'_{ii} &= c^2 a_{ii} + s^2 a_{jj} - 2sca_{ij} \\
a'_{jj} &= s^2 a_{ii} + c^2 a_{jj} + 2sca_{ij} \\
a'_{ij} &= (c^2 - s^2)a_{ij} + sc(a_{ii} - a_{jj}) \Rightarrow a'_{ij} = 0
\end{aligned}$$

Note: A' is always symmetric.

1.3 The Eigengap Heuristic

In order to determine the number of clusters k, we will use eigengap heuristic as follow:
let $(\delta_i)_{i=1,\ldots,n-1}$ be the eigengap for the increasingly ordered eigenvalues $0 \leq \lambda_1 \leq \ldots \leq \lambda_n$ of L_{norm}, defined as:

$$\delta_i = |\lambda_i - \lambda_{i+1}|$$

The eigengap measure could indicate the number of clusters k through the following estimation:

$$k = argmax_i(\delta_i), \quad i = 1, \ldots, \left\lfloor \frac{n}{2} \right\rfloor$$

In case of equality in the $argmax$ of some eigengaps, use the lowest index.

2 Assignment Description

Implement the following files:

1. `spkmeans.py`: Python interface of your code.
2. `spkmeans.h`: C header file.
3. `spkmeans.c`: C interface of your code.
4. `spkmeansmodule.c`: Python C API wrapper.
5. `setup.py`: The setup file.
6. `comp.sh`: Your compilation script.
7. `*.c/h`: Other modules and headers per your design.

2.1 Python Program (`spkmeans.py`)

1. Reading user CMD arguments:
   (a) `k` (int, $< N$): Number of required clusters. If equal 0, use the heuristic 1.3.
   (b) `goal` (enum): Can get the following values:
      i. `spk`: Perform full spectral kmeans as described in 1.
      ii. `wam`: Calculate and output the Weighted Adjacency Matrix as described in 1.1.1.
      iii. `ddg`: Calculate and output the Diagonal Degree Matrix as described in 1.1.2.
      iv. `lnorm`: Calculate and output the Normalized Graph Laplacian as described in 1.1.3.
      v. `jacobi`: Calculate and output the eigenvalues and eigenvectors as described in 1.2.1.
   (c) `file_name` (.txt or .csv): The path to the file that will contain N observations, the file extension is .txt or .csv.
2. Implementation of the k-means++ algorithm when the `goal=spk`, as detailed in HW2:
   (a) Set `np.random.seed(0)` **at the beginning of your code**.
   (b) Use `np.random.choice()` for random selection.
3. Interfacing with your C extension `spkmeansmodule`. All implementations of the different goals must be performed by calling the C extension.
4. Outputting the following:
   In case of 'spk':
   - The first line will be the indices of the observations chosen by the K-means++ algorithm as the initial centroids. We refer to the first observation index as 0, the second as 1 and so on, up until N - 1.
   - The second line onwards will be the calculated final centroids from the K-means algorithm, separated by a comma, such that each centroid is in a line of its own.

   For the other cases, output the required matrix separated by a comma, such that each row is in a line of its own.

Example:

```
>>>python3 spkmeans.py 3 spk input_1.txt
0,4,22
-4.2435,9.1568,5.4105
3.3226,-1.3896,-9.1927
8.2239,-8.5714,-8.4985
```

2.2 C Program (`spkmeans.c`)

This is the C implementation program, with the following requirements:

1. Reading user CMD arguments:
   (a) `goal` (enum): Can get the following values:
      i. `wam`: Calculate and output the Weighted Adjacency Matrix as described in 1.1.1.
      ii. `ddg`: Calculate and output the Diagonal Degree Matrix as described in 1.1.2.
      iii. `lnorm`: Calculate and output the Normalized Graph Laplacian as described in 1.1.3.
      iv. `jacobi`: Calculate and output the eigenvalues and eigenvectors as described in 1.2.1.
   (b) `file_name`: The path to the file that will contain N observations, the file extension is .txt or .csv.
2. Outputting the following:
   Output the required matrix separated by a comma, such that each row is in a line of its own.

The program must compile cleanly (no errors, no warnings) when running the following command:

```
$bash comp.sh
```

After successful compilation the program can run for Example:

```
>>./spkmeans 3 lnorm input_1.txt
-4.2435,9.1568,5.4105
3.3226,-1.3896,-9.1927
8.2239,-8.5714,-8.4985
```

2.3 Python C API (`spkmeansmodule.c`)

In this file you will define your C extension which will be, mainly, your wrap of the algorithm implemented in `spkmeans.c`.

You can use functions implemented only in `spkmeans.c`. It is up to you to decide which and what type of arguments you want to pass when calling the API. Same thing about the return.

2.4 C Header file(`spkmeans.h`)

This header have to define all functions prototypes that is being used in `spkmeansmodule.c` and implemented at `spkmeans.c`.

2.5 Setup (`setup.py`)

This is the build used to create the *.so file that will allow `spkmeans.py` to import `spkmeans`.

2.6 Build and Running

1. The extension must built cleanly (no errors, no warnings) when running the following command:

```
$python setup.py build_ext --inplace
```

2.7 `comp.sh`

Script for compiling your files. The compilation command should include all the flags as below with your modules accordingly:

```
#!/bin/bash
# Script to compile and execute a c program
gcc -ansi -Wall -Wextra -Werror -pedantic-errors spkmeans.c -lm -o spkmeans
```

2.8 Assumptions and requirements:

1. You may assume that the input files are in the correct format.
2. All the algorithmic implementations detailed at 1 must be implemented in the C program, except the k-means++ which is in Python (as in HW2).
3. Validate that the command line arguments are in correct format.
4. Outputs must be formatted to 4 decimal places (use: '%.4f') in both languages, for example:
   - 8.88885 ⇒ 8.8888
   - 5.92237098749999997906 ⇒ 5.9224
   - 2.231 ⇒ 2.2310
5. If the Jacobi doesn't reach convergence 5, you should run it up to 100 iteration.
6. Your code should be gracefully partitioned into functions. The design of the program, including the main interface\integration, function declarations, and how they interact is entirely up to you. You should aim for modularity, reuse of code, clarity, and logical partition.
7. Source files should be commented at critical points in the code and at function declarations.
8. Please avoid long lines of code, and shorten lines that exceed 80 characters.
9. You may import external includes (in C) or modules (in Python) that are not mentioned in this document.
10. Comment your code. If you use code from the internet, please cite the source.
11. Handle errors as following:
    (a) In case of invalid input, print "Invalid Input!" and terminate the program.

(b) Else, print "An Error Has Occurred" and terminate.

12. Do not forget to free any memory you allocated.
13. You can assume that data dimensionality is up to 1000 points and up to 10 features.
14. You can assume that all given data points are different.
15. Use `double` in C and `float` in Python for all vector's elements.
16. There is an example doc that depicts the steps of the algorithm (you can find at Moodle under the project section), you can generate data using the provided method in the example document and test yourself.

2.9 Submission

1. Please submit a file named `id1_id2_final.zip` via Moodle, where id1 and id2 are the ids of the partners.
   (a) In case of individual submission, id2 must be 111111111
   (b) Put the following files in a folder called `id1_id2_final`:
      i. `spkmeans.py`
      ii. `spkmeans.h`
      iii. `spkmeans.c`
      iv. `spkmeansmodule.c`
      v. `setup.py`
      vi. `comp.sh`
      vii. `*.c/h`
   (c) Zip the folder using the following linux cmd:

```
$zip -r id1_id2_final.zip id1_id2_final
```

   Do not use other ways to create the zip!

2.10 Remarks

1. For any question regarding the assignment, please post at the Final_Project's discussion forum.
2. No test files are provided, you can generate data using `sklearn.datasets.make_blobs`
3. An example of using C code for direct call and for Python is provided with the assignment under `Demo1`.
4. You can create symmetric matrices in order to test Jacobi by using:

```
a = np.random.rand(N, N)
m = np.tril(a) + np.tril(a, -1).T
```

References

[1] Andrew Ng, Michael Jordan, and Yair Weiss. On spectral clustering: Analysis and an algorithm. *Advances in neural information processing systems*, 14:849–856, 2001.

[2] Ulrike Von Luxburg. A tutorial on spectral clustering. *Statistics and computing*, 17(4):395–416, 2007.