HW6

May 15, 2020

1 CS 168 Spring Assignment 6

SUNet ID(s): 05794739

Name(s): Luis A. Perez

Collaborators: None

By turning in this assignment, I agree by the Stanford honor code and declare that all of this is my own work.

2 Imports

[28]:
```
import collections
import matplotlib.pyplot as plt
import scipy

import numpy as np
from PIL import Image
from sklearn import decomposition
import pandas as pd
import seaborn as sns
import os
import warnings

from typing import Dict, List, Text, Tuple

# Make figure larger
plt.rcParams['figure.figsize'] = [10, 5]

# Set numpy seed for consistent results.
np.random.seed(1)
```

[29]:
```
class Globals:
    DATA_PATH = 'data/'
```

3 Part 1

3.1 Part 1b

[30]:
```
def line_graph(n=100):
    """Returns degree and adjacency matrices for line graph."""
    D = 2 * np.ones(n)
    D[0] = 1
    D[n - 1] = 1
    D = np.diag(D)
    A = np.zeros((n, n))
    for i in range(n-1):
        A[i, i + 1] = 1
        A[i + 1, i] = 1
    return A, D

def circle_graph(n=100):
    """Returns degree and adjacency matrices for circle graph."""
    D = 2 * np.ones(n)
    D = np.diag(D)
    A = np.zeros((n, n))
    for i in range(n):
        A[i, (i + 1) % n] = 1
        A[(i + 1) % n, i] = 1
    return A, D

def line_graph_plus_one(n=100):
    """
    Returns degree and adjacency matrices for an n-1
    point line graph with a final point connect to all previous points.
    """
    D = 3 * np.ones(n)
    D[0] = 2
    D[n - 2] = 2
    D[n - 1] = n - 1
    D = np.diag(D)
    A = np.zeros((n, n))
    for i in range(n-2):
        A[i, i + 1] = 1
        A[i + 1, i] = 1
    for i in range(n-1):
        A[n-1, i] = 1
        A[i, n-1] = 1
    return A, D

def circle_graph_plus_one(n=100):
    """
    Returns degree and adjacency matrices for an n-1 circle graph
```

```
    with a final point connecting to all previous points.
    """
    D = 3 * np.ones(n)
    D[n-1] = n-1
    D = np.diag(D)
    A = np.zeros((n, n))
    for i in range(n-1):
        A[i, i + 1] = 1
        A[i + 1, i] = 1
        A[i, n - 1] = 1
        A[n - 1, i] = 1
    return A, D
```

[31]:
```
def compute_eigens(graph_type, n=100):
    A, D = globals()[graph_type](n=n)
    L = D - A
    return {
        'L' : np.linalg.eig(L),
        'A' :np.linalg.eig(A)
    }
```

[32]:
```
x = compute_eigens('circle_graph')['L'][1][:, np.
↪argsort(compute_eigens('circle_graph')['L'][0])][:, 0]
```

[33]:
```
def problem1b():
    for graph_type in ['circle_graph', 'circle_graph_plus_one', 'line_graph',␣
↪'line_graph_plus_one']:
        for typ, (eig_val, eig_v) in compute_eigens(graph_type).items():
            idx = np.argsort(eig_val) # From small to large
            sorted_v = eig_v[:, idx] # Sort by eigenvalue.
            for name, i in zip(['smallest', 'smaller', 'larger', 'largest'],␣
↪[0, 1, -2, -1]):
                v = sorted_v[:, i]
                plt.ylim((-0.25, 0.25))
                plt.plot(range(len(v)), v, label=name)
            plt.legend()
            plt.xlabel('Vector Index')
            plt.ylabel('Vector Value')
            plt.title(f'{name.capitalize()} Eigenvectors of matrix {typ} of␣
↪{graph_type}.')
            plt.savefig(f'figures/eigenvectors_{graph_type}_{typ}.png',␣
↪format='png')
            plt.close()
```

[34]:
```
problem1b()
```

3.2 Problem 1c

[35]:
```
def edges(graph_type, n=100):
    """Returns an iterator over the edges (i,j) in the given graph."""
    _EDGES = {
        'line_graph': zip(range(0,n-1), range(1,n)),
        'circle_graph': zip(range(0,n), list(range(1,n)) + [0]),
        'line_graph_plus_one': list(zip(range(0,n-2), range(1,n-1))) +␣
↪list(zip(range(n-2), [n-1] * (n-1))),
        'circle_graph_plus_one': list(zip(range(0,n), list(range(1,n)) + [0]))␣
↪+ list(zip(range(n-2), [n-1] * (n-1))),
    }
    return _EDGES[graph_type]
```

[36]:
```
def ploblem1c():
    for graph_type in ['circle_graph', 'circle_graph_plus_one', 'line_graph',␣
↪'line_graph_plus_one']:
        eig_val, eig_v = compute_eigens(graph_type)['L']
        idx = np.argsort(eig_val)
        sorted_v = eig_v[:, idx]
        v2, v3 = sorted_v[:, 1], sorted_v[:, 2]
        plt.axis('equal')
        plt.scatter(v2, v3)
        for i,j in edges(graph_type):
            plt.plot([v2[i], v2[j]], [v3[i], v3[j]])
        plt.title(f'Eigenvector Project of {graph_type}')
        plt.xlabel('v2 - 2nd smallest eigenvector')
        plt.ylabel('v3 - 3rd smallest eigenvector')
        plt.savefig(f'figures/eignevector_projection_{graph_type}.png',␣
↪format='png')
        plt.close()
```

[37]:
```
ploblem1c()
```

3.3 Problem 1d

[38]:
```
def l2_dist(X, Y):
    """Computes the L2 pairwise distance between all elements in X,Y.

    Args:
        X: An (n,k) matrix where each row is an element.
        Y: An (m,k) matrix where each row is an element.

    Returns:
        D: An (n,m) matrix where D[i][j] is the distance L2 distance
            between X[i,:] and Y[j,:].
    """
```

```
    (n,k1), (m, k2) = np.shape(X), np.shape(Y)
    assert k1 == k2
    k = k1
    X2 = np.diag(np.dot(X, X.T)).reshape((n, 1))
    Y2 = np.diag(np.dot(Y, Y.T)).reshape((1, m))
    XY = np.dot(X, Y.T)
    return np.sqrt(X2 + Y2 - 2*XY)
```

[39]:
```
def problem1d():
    points = np.random.uniform(size=(500, 2))
    distances = l2_dist(points, points)
    A = (distances < 0.25).astype(int)
    assert np.allclose(A, A.T) # Matrix is symetric.
    D = np.diag(np.sum(A, axis=1))
    L = D - A
    val, eigs = np.linalg.eig(L)
    idx = np.argsort(val)
    v2, v3 = eigs[:, idx[1]], eigs[:, idx[2]]
    plt.scatter(v2, v3)

    mask = (points[:, 0] < 0.5) & (points[:, 1] < 0.5)
    plt.scatter(v2[mask], v3[mask])
    plt.savefig('figures/cluster_random.png', format='png')
    plt.title('Projection of Random Graph with Neighborhood Defined by L2 Dist␣
↪< 0.25')
    plt.xlabel('v2 - second smallest eigenvector of laplacian')
    plt.ylabel('v3 - third smallest eigenvector of laplacian')
    plt.close()
```

[40]:
```
problem1d()
```

4 Problem 2

4.1 Problem 2a

[14]:
```
def read_friendship_graph():
    """Returns Adjacency matrix of friendship graph."""
    data = pd.read_csv(os.path.join(Globals.DATA_PATH, 'cs168mp6.csv'),␣
↪header=None, names=['a', 'b'])
    assert len(data) == 61796
    assert len(set(data.a.unique()) | set(data.b.unique())) == 1495
    n = 1495
    A = np.zeros((n, n))
    for _, row in data.iterrows():
        A[row.a - 1, row.b - 1] = 1
        A[row.b - 1, row.a - 1] = 1
```

```
    return A.astype(int)
```

4.2 Problem 2b

[25]:
```
def problem2b():
    A = read_friendship_graph()
    D = np.diag(np.sum(A, axis=1))
    L = D - A
    vals, _ = np.linalg.eig(L)
    print('Smallest 12 eigenvalues, in order:')
    for val in sorted(vals)[:12]:
        print(f'{val:.3f}')
```

[26]:
```
problem2b()
```

```
Smallest 12 eigenvalues, in order:
-0.000
-0.000
-0.000
-0.000
-0.000
0.000
0.014
0.054
0.074
0.081
0.120
0.133
```

4.3 Problem 2c

[446]:
```
def problem2c():
    A = read_friendship_graph()
    D = np.diag(np.sum(A, axis=1))
    L = D - A
    vals, eighs = np.linalg.eig(L)
    sorted_eighs = eighs[:, np.argsort(vals)]
    return sorted_eighs
```

[447]:
```
eigs = problem2c()
```

[473]:
```
# Try a couple at different rounding factors until you get 6 distinct clusters.
for i in range(6):
    rounded = np.round(eigs[:, i], decimals=12)
    print(f'Number of components {len(np.unique(rounded))}')
    print(f'Component sizes {np.unique(rounded, return_counts=True)[1]}')
```

```
Number of components 6
Component sizes [   2    2    2    3 1484    2]
Number of components 6
Component sizes [   3    2 1484    2    2    2]
Number of components 6
Component sizes [   2    2    3 1484    2    2]
Number of components 6
Component sizes [   2    2    3 1484    2    2]
Number of components 6
Component sizes [1484    2    2    2    3    2]
Number of components 6
Component sizes [   2 1484    2    2    3    2]
```

4.4 Problem 2d

[47]:
```python
def conductance(S, A):
    n = A.shape[0]
    complementS = set(range(n)) - S
    num = 0
    for i in S:
        for j in complementS:
            num += A[i][j]
    AS = np.sum(A[list(S), :])
    AnotS = np.sum(A[list(complementS), :])
    return num / min(AS, AnotS)
```

[263]:
```python
def search_for_clusters(eigs, condutance_t=0.1, min_size=150, max_size=300,␣
↪num_unique_to_return=1):
    found = []
    prev_sets = []
    for v_idx in range(6, 40):
        print(f'Searching {v_idx}')
        for t in np.arange(0, 0.08, 0.00002):
            indexes = (np.abs(eigs[:, v_idx]) < t)
            vals = eigs[:, v_idx][indexes]
            if len(vals) >= min_size and len(vals) <= max_size:
                cond = conductance(set(indexes.nonzero()[0]), A)
                if cond < condutance_t:
                    if not prev_sets:
                        found.append((t, v_idx, cond))
                        prev_sets.append(set(indexes.nonzero()[0]))
                    else:
                        s = set(indexes.nonzero()[0])
                        skip = False
                        for prev in prev_sets:
                            if len(s & prev) > 10:
                                skip = True
```

```
                                break
                        if not skip:
                            found.append((t, v_idx, cond))
                            prev_sets.append(s)
                    if len(found) >= num_unique_to_return:
                        return found
    return found
```

[330]:
```
# Read the graph.
A = read_friendship_graph()
D = np.diag(np.sum(A, axis=1))
L = D - A
vals, eigs = np.linalg.eig(L)
idx = np.argsort(vals)
sorted_eigs = eigs[:, idx]
```

[331]:
```
# Search to find a candidate eigenvector.
search_for_clusters(sorted_eigs, num_unique_to_return=1)
```

```
Searching 6
Searching 7
Searching 8
```

[331]:
```
[(0.0002, 8, 0.08864468864468865)]
```

[342]:
```
# Try looking at different values here to identify closely clustered sets of␣
 ↪points.
s1 = set((np.abs(sorted_eigs[:, 8]) < 0.0002).nonzero()[0])
len(s1), conductance(s1, A)
```

[342]:
```
(229, 0.08864468864468865)
```

[343]:
```
s2 = set(((sorted_eigs[:, 8] > 0.0002) & (sorted_eigs[:, 8] < 0.0045)).
 ↪nonzero()[0])
len(s2), conductance(s2, A)
```

[343]:
```
(561, 0.05860035888233786)
```

[344]:
```
s3 = set(((sorted_eigs[:, 8] > 0.0045) & (sorted_eigs[:, 8] < 0.00488)).
 ↪nonzero()[0])
len(s3), conductance(s3, A)
```

[344]:
```
(362, 0.08672401767030923)
```

[345]:
```
# Assert the ids are disjoint.
s1 & s2, s2 & s3, s1 & s3
```

[345]:
```
(set(), set(), set())
```

[351]:
```
np.array(list(s1))[:10] + 1
```

[351]:
```
array([   2,    6,  519,    8,  521,  522, 1032, 1036,   13,  527])
```

[352]:
```
np.array(list(s2))[:10] + 1
```

[352]:
```
array([ 1,  3,  4,  9, 10, 11, 14, 16, 19, 20])
```

[353]:
```
np.array(list(s3))[:10] + 1
```

[353]:
```
array([ 5,  7, 23, 27, 30, 42, 44, 46, 47, 50])
```

[354]:
```
plt.title('Eigenvector corresponding to 9-th smallest eigenvalue.')
plt.xlabel('Vector Index')
plt.ylabel('Vector Value')
plt.plot(range(sorted_eigs.shape[0]), sorted_eigs[:, 8])
plt.savefig('figures/eigenvector_9.png', format='png')
plt.close()
```

4.5 Problem 2e

[355]:
```
def problem2e():
    A = read_friendship_graph()
    n = A.shape[0]
    selected = set(np.random.choice(range(n), size=150, replace=False))
    cond = conductance(selected, A)
    print(f'The conductance of the randomly selected set of {150} nodes is␣
 ↪{cond:.4f}.')
```

[356]:
```
problem2e()
```

```
The conductance of the randomly selected set of 150 nodes is 0.8975.
```

[ ]:
```

```