In [1]:
```
from typing import List

import numpy as np
import scipy.sparse as sp
```

Project 4: Spectral clustering users based on their preferences (50 pt)

The goal of this task is to find groups of users with similar preferences using **Spectral clustering**. You are given a fragment of the Yelp social network, represented by an undirected weighted graph. Nodes in the graph represent users. If two users are connected by an edge of weight w, it means that they have both left positive reviews to the same w restaurants.

Additionally, you are given a matrix `F` that encodes user preferences to different categories of restaurants. If `F[i, c] = 1`, then user `i` likes restaurants in category `c`.

You are allowed to use the imported functions (`eigsh`, `KMeans`, `normalize`).

General remarks

Do not add or modify any code outside of the following comment blocks, or where otherwise explicitly stated.

```
############################################################
# YOUR CODE HERE
...
############################################################
```

After you fill in all the missing code, restart the kernel and re-run all the cells in the notebook.

The following things are **NOT** allowed:

- Using additional `import` statements
- Copying / reusing code from other sources (e.g. code by other students)

If you plagiarise even for a single project task, you won't be eligible for the bonus this semester.

Load the data

- `N` = number of users (nodes in the graph)
- `C` = number of categories
- The graph is stored as a *sparse adjacency matrix* `A` (shape `[N, N]`).
- User preferences are stored in a *feature matrix* `F` (shape `[N, C]`). They will only be used for the final part of the assignment (Part 3)
- Name of each category is provided in the list `categories` (length `[C]`).

In [2]:
```
A = sp.load_npz('A.npz')
F = np.load('F.npy')
categories = np.load('categories.npy', allow_pickle=True).tolist()
```

In [3]:
```
assert A.shape[0] == F.shape[0]
assert F.shape[1] == len(categories)
```

In [4]:
```
print(f'The adjacency matrix is {"symmetric" if (A != A.T).sum() == 0 else "
asymmetric"}')
```

```
The adjacency matrix is symmetric
```

1. Implementing spectral clustering (35 pt)

1.1. Construct the graph Laplacian (10 pt)

First, we need to construct the Laplacian for the given graph (*Do only use sparse operations, see Scipy Sparse (https://docs.scipy.org/doc/scipy/reference/sparse.html)*).

Given the **adjacency matrix** $A \in \mathbb{R}^{N \times N}$, we define the **degree matrix** $D \in \mathbb{R}^{N \times N}$ of an undirected graph as

$$D_{ij} = \begin{cases} \sum_{k=1}^{N} A_{ik} & if \ \ i = j \\ 0 & if \ \ i \neq j \end{cases}$$

If our goal is to minimize the **ratio cut**, we will need to use the **unnormalized Laplacian**, defined as

$$L_{unnorm} = D - A.$$

If our goal is to minimze the **normalized cut**, we will need to use the **normalized Laplacian** (a.k.a. symmetrized Laplacian), defined as

$$L_{sym} = I - D^{-1/2} A D^{-1/2}$$

In [5]:
```
def construct_laplacian(A: sp.csr_matrix,
                        norm_laplacian: bool) -> sp.csr_matrix:
    """Construct Laplacian of a graph.

    Parameters
    ----------
    A : scipy.sparse.csr_matrix, shape [N, N]
        Adjacency matrix of the graph.
    norm_laplacian : bool
        Whether to construct the normalized graph Laplacian or not.
        If True, construct the normalized (symmetrized) Laplacian, L = I - D
^{-1/2} A D^{-1/2}.
        If False, construct the unnormalized Laplacian, L = D - A.

    Returns
    -------
    L : scipy.sparse.csr_matrix, shape [N, N]
        Laplacian of the graph.

    """
    ###############################################################
    # YOUR CODE HERE
    N = A.shape[0]

    # sum up each row
    D_vec = np.array(A.sum(axis=1)).squeeze()

    if norm_laplacian:
        D = sp.diags(np.power(D_vec, -1 / 2))
        L = sp.identity(N) - D * A * D
    else:
        D = sp.diags(D_vec)
        L = D - A
    ###############################################################
    return L
```

In [ ]:

1.2. Spectral embedding (10 pt)

Now, we have to compute the spectral embedding for the given graph.

In order to partition the graph into k clusters, such that the desired cut (ratio or normalized) is minimized, we need to consider the k eigenvectors corresponding to the k smallest eigenvalues of the graph Laplacian.

Since the Laplacian matrix is sparse and symmetric, we can use the function `eigsh` from the `scipy.sparse.linalg` package in order to find eigendecomposition of L (`eig` - eigendecomposition, `s` - sparse, `h` - Hermitian). The function `eigsh` directly allows you to find the smallest / largest eigenvalues by specifying the `k` and `which` parameters.

Keep in mind that the Laplacian matrix is always positive semi-definite when picking the appropriate value for the `which` parameter.

In [6]:
```
from scipy.sparse.linalg import eigsh
```

In [7]:
```
help(eigsh)
```

```
Help on function eigsh in module scipy.sparse.linalg.eigen.arpack.arpack:

eigsh(A, k=6, M=None, sigma=None, which='LM', v0=None, ncv=None, maxiter=Non
e, tol=0, return_eigenvectors=True, Minv=None, OPinv=None, mode='normal')
    Find k eigenvalues and eigenvectors of the real symmetric square matrix
    or complex hermitian matrix A.

    Solves ``A * x[i] = w[i] * x[i]``, the standard eigenvalue problem for
    w[i] eigenvalues with corresponding eigenvectors x[i].

    If M is specified, solves ``A * x[i] = w[i] * M * x[i]``, the
    generalized eigenvalue problem for w[i] eigenvalues
    with corresponding eigenvectors x[i].

    Parameters
    ----------
    A : ndarray, sparse matrix or LinearOperator
        A square operator representing the operation ``A * x``, where ``A`` i
s
        real symmetric or complex hermitian. For buckling mode (see below)
        ``A`` must additionally be positive-definite.
    k : int, optional
        The number of eigenvalues and eigenvectors desired.
        `k` must be smaller than N. It is not possible to compute all
        eigenvectors of a matrix.

    Returns
    -------
    w : array
        Array of k eigenvalues.
    v : array
        An array representing the `k` eigenvectors.  The column ``v[:, i]`` i
s
        the eigenvector corresponding to the eigenvalue ``w[i]``.

    Other Parameters
    ----------------
    M : An N x N matrix, array, sparse matrix, or linear operator representin
g
        the operation ``M @ x`` for the generalized eigenvalue problem

            A @ x = w * M @ x.

        M must represent a real, symmetric matrix if A is real, and must
        represent a complex, hermitian matrix if A is complex. For best
        results, the data type of M should be the same as that of A.
        Additionally:

            If sigma is None, M is symmetric positive definite.

            If sigma is specified, M is symmetric positive semi-definite.

            In buckling mode, M is symmetric indefinite.

        If sigma is None, eigsh requires an operator to compute the solution
        of the linear equation ``M @ x = b``. This is done internally via a
        (sparse) LU decomposition for an explicit matrix M, or via an
        iterative solver for a general linear operator.  Alternatively,
        the user can supply the matrix or operator Minv, which gives
        ``x = Minv @ b = M^-1 @ b``.
    sigma : real
        Find eigenvalues near sigma using shift-invert mode.  This requires
        an operator to compute the solution of the linear system
        ``[A - sigma * M] x = b``, where M is the identity matrix if
        unspecified.  This is computed internally via a (sparse) LU
        decomposition for explicit matrices A & M, or via an iterative
        solver if either A or M is a general linear operator.
        Alternatively, the user can supply the matrix or operator OPinv,
```

```
            which gives ``x = OPinv @ b = [A - sigma * M]^-1 @ b``.
            Note that when sigma is specified, the keyword 'which' refers to
            the shifted eigenvalues ``w'[i]`` where:

                if mode == 'normal', ``w'[i] = 1 / (w[i] - sigma)``.

                if mode == 'cayley', ``w'[i] = (w[i] + sigma) / (w[i] - sigma)``.

                if mode == 'buckling', ``w'[i] = w[i] / (w[i] - sigma)``.

            (see further discussion in 'mode' below)
        v0 : ndarray, optional
            Starting vector for iteration.
            Default: random
        ncv : int, optional
            The number of Lanczos vectors generated ncv must be greater than k an
    d
            smaller than n; it is recommended that ``ncv > 2*k``.
            Default: ``min(n, max(2*k + 1, 20))``
        which : str ['LM' | 'SM' | 'LA' | 'SA' | 'BE']
            If A is a complex hermitian matrix, 'BE' is invalid.
            Which `k` eigenvectors and eigenvalues to find:

                'LM' : Largest (in magnitude) eigenvalues.

                'SM' : Smallest (in magnitude) eigenvalues.

                'LA' : Largest (algebraic) eigenvalues.

                'SA' : Smallest (algebraic) eigenvalues.

                'BE' : Half (k/2) from each end of the spectrum.

            When k is odd, return one more (k/2+1) from the high end.
            When sigma != None, 'which' refers to the shifted eigenvalues ``w'[i]
    ``
            (see discussion in 'sigma', above).  ARPACK is generally better
            at finding large values than small values.  If small eigenvalues are
            desired, consider using shift-invert mode for better performance.
        maxiter : int, optional
            Maximum number of Arnoldi update iterations allowed.
            Default: ``n*10``
        tol : float
            Relative accuracy for eigenvalues (stopping criterion).
            The default value of 0 implies machine precision.
        Minv : N x N matrix, array, sparse matrix, or LinearOperator
            See notes in M, above.
        OPinv : N x N matrix, array, sparse matrix, or LinearOperator
            See notes in sigma, above.
        return_eigenvectors : bool
            Return eigenvectors (True) in addition to eigenvalues.
            This value determines the order in which eigenvalues are sorted.
            The sort order is also dependent on the `which` variable.

                For which = 'LM' or 'SA':
                    If `return_eigenvectors` is True, eigenvalues are sorted by
                    algebraic value.

                    If `return_eigenvectors` is False, eigenvalues are sorted by
                    absolute value.

                For which = 'BE' or 'LA':
                    eigenvalues are always sorted by algebraic value.

                For which = 'SM':
                    If `return_eigenvectors` is True, eigenvalues are sorted by
                    algebraic value.
```

```
            If `return_eigenvectors` is False, eigenvalues are sorted by
            decreasing absolute value.

    mode : string ['normal' | 'buckling' | 'cayley']
        Specify strategy to use for shift-invert mode.  This argument applies
        only for real-valued A and sigma != None.  For shift-invert mode,
        ARPACK internally solves the eigenvalue problem
        ``OP * x'[i] = w'[i] * B * x'[i]``
        and transforms the resulting Ritz vectors x'[i] and Ritz values w'[i]
        into the desired eigenvectors and eigenvalues of the problem
        ``A * x[i] = w[i] * M * x[i]``.
        The modes are as follows:

            'normal' :
                OP = [A - sigma * M]^-1 @ M,
                B = M,
                w'[i] = 1 / (w[i] - sigma)

            'buckling' :
                OP = [A - sigma * M]^-1 @ A,
                B = A,
                w'[i] = w[i] / (w[i] - sigma)

            'cayley' :
                OP = [A - sigma * M]^-1 @ [A + sigma * M],
                B = M,
                w'[i] = (w[i] + sigma) / (w[i] - sigma)

        The choice of mode will affect which eigenvalues are selected by
        the keyword 'which', and can also impact the stability of
        convergence (see [2] for a discussion).

    Raises
    ------
    ArpackNoConvergence
        When the requested convergence is not obtained.

        The currently converged eigenvalues and eigenvectors can be found
        as ``eigenvalues`` and ``eigenvectors`` attributes of the exception
        object.

    See Also
    --------
    eigs : eigenvalues and eigenvectors for a general (nonsymmetric) matrix A
    svds : singular value decomposition for a matrix A

    Notes
    -----
    This function is a wrapper to the ARPACK [1]_ SSEUPD and DSEUPD
    functions which use the Implicitly Restarted Lanczos Method to
    find the eigenvalues and eigenvectors [2]_.

    References
    ----------
    .. [1] ARPACK Software, http://www.caam.rice.edu/software/ARPACK/
    .. [2] R. B. Lehoucq, D. C. Sorensen, and C. Yang,  ARPACK USERS GUIDE:
       Solution of Large Scale Eigenvalue Problems by Implicitly Restarted
       Arnoldi Methods. SIAM, Philadelphia, PA, 1998.

    Examples
    --------
    >>> from scipy.sparse.linalg import eigsh
    >>> identity = np.eye(13)
    >>> eigenvalues, eigenvectors = eigsh(identity, k=6)
    >>> eigenvalues
    array([1., 1., 1., 1., 1., 1.])
    >>> eigenvectors.shape
    (13, 6)
```

In [8]:
```
def spectral_embedding(A: sp.csr_matrix, num_clusters: int,
                       norm_laplacian: bool) -> np.array:
    """Compute spectral embedding of nodes in the given graph.

    Parameters
    ----------
    A : scipy.sparse.csr_matrix, shape [N, N]
        Adjacency matrix of the graph.
    num_clusters : int
        Number of clusters to detect in the data.
    norm_laplacian : bool, default False
        Whether to use the normalized graph Laplacian or not.

    Returns
    -------
    embedding : np.array, shape [N, num_clusters]
        Spectral embedding for the given graph.
        Each row represents the spectral embedding of a given node.

    """
    if (A != A.T).sum() != 0:
        raise ValueError(
            "Spectral embedding doesn't work if the adjacency matrix is not
symmetric."
        )
    if num_clusters < 2:
        raise ValueError("The clustering requires at least two clusters.")
    if num_clusters > A.shape[0]:
        raise ValueError(
            f"We can have at most {A.shape[0]} clusters (number of nodes).")
    ###########################################################
    # YOUR CODE HERE
    L = construct_laplacian(A, norm_laplacian)
    eigenvalues, eigenvectors = eigsh(L, k=num_clusters, which='SM')
    ###########################################################
    return eigenvectors
```

1.3. Determine the clusters based on the spectral embedding (15 pt)

You should use the K-means algorithm for assigning nodes to clusters, once the spectral embedding is computed.

One thing you should keep in mind, is that when using the **normalized Laplacian**, the rows of the embedding matrix **have to** be normalized to have unit L_2 norm.

In [9]:
```
from sklearn.cluster import KMeans
from sklearn.preprocessing import normalize
```

In [10]:
```
def spectral_clustering(A: sp.csr_matrix, num_clusters: int, norm_laplacian:
bool, seed: int = 42) -> np.array:
    """Perform spectral clustering on the given graph.

    Parameters
    ----------
    A : scipy.sparse.csr_matrix, shape [N, N]
        Adjacency matrix of the graph.
    num_clusters : int
        Number of clusters to detect in the data.
    norm_laplacian : bool, default False
        Whether to use the normalized graph Laplacian or not.
    seed : int, default 42
        Random seed to use for the `KMeans` clustering.

    Returns
    -------
    z_pred : np.array, shape [N]
        Predicted cluster indicators for each node.

    """
    model = KMeans(num_clusters, random_state=seed)
    ############################################################
    # YOUR CODE HERE
    embedding = spectral_embedding(A, num_clusters, norm_laplacian)

    if norm_laplacian :
        embedding = normalize(embedding, norm='l2', axis=1)

    z_pred = model.fit_predict(embedding)

    ############################################################
    return z_pred
```

2. Quantitatively evaluate the results (10 pt)

In [11]:
```
def labels_to_list_of_clusters(z: np.array) -> List[List[int]]:
    """Convert predicted label vector to a list of clusters in the graph.
    This function is already implemented, nothing to do here.

    Parameters
    ----------
    z : np.array, shape [N]
        Predicted labels.

    Returns
    -------
    list_of_clusters : list of lists
        Each list contains ids of nodes that belong to the same cluster.
        Each node may appear in one and only one partition.

    Examples
    --------
    >>> z = np.array([0, 0, 1, 1, 0])
    >>> labels_to_list_of_clusters(z)
    [[0, 1, 4], [2, 3]]

    """
    return [np.where(z == c)[0] for c in np.unique(z)]
```

2.1. Compute ratio cut (5 pt)

Your task is to implement functions for computing the **ratio cut** and **normalized cut** for a given partition.

Ratio cut and normalized cut are defined on the slide 14 of the lecture slides.

The function `labels_to_list_of_clusters` can be helpful here.

In [12]:
```
def compute_ratio_cut(A: sp.csr_matrix, z: np.array) -> float:
    """Compute the ratio cut for the given partition of the graph.

    Parameters
    ----------
    A : scipy.sparse.csr_matrix, shape [N, N]
        Adjacency matrix of the graph.
    z : np.array, shape [N]
        Cluster indicators for each node.

    Returns
    -------
    ratio_cut : float
        Value of the cut for the given partition of the graph.

    """

    ############################################################
    # YOUR CODE HERE

    N = A.shape[0]
    label_list = labels_to_list_of_clusters(z)

    list_all = list(range(N))
    ratio_cut = 0
    for cat in label_list:
        cut = 0
        num_user = len(cat)
        indices = list(set(list_all).difference(set(cat)))
        for user in cat:
            row = A[user,:].toarray().squeeze()
            cut += row[indices].sum()
        cut = cut/num_user
        ratio_cut += cut

    ############################################################
    return ratio_cut
```

2.2. Compute normalized cut (5 pt)

Important: if a cluster only contains a single node, define its volume to be 1 to avoid division by zero errors.

In [13]:
```
def compute_normalized_cut(A: sp.csr_matrix, z: np.array) -> float:
    """Compute the normalized cut for the given partition of the graph.

    Parameters
    ----------
    A : scipy.sparse.csr_matrix, shape [N, N]
        Adjacency matrix of the graph.
    z : np.array, shape [N]
        Cluster indicators for each node.

    Returns
    -------
    norm_cut : float
        Value of the normalized cut for the given partition of the graph.

    """

    ###########################################################
    # YOUR CODE HERE
    N = A.shape[0]
    label_list = labels_to_list_of_clusters(z)
    list_all = list(range(N))
    norm_cut = 0
    for cat in label_list:
        cut = 0
        vol = 0
        num_user = len(cat)

        indices = list(set(list_all).difference(set(cat)))
        for user in cat:
            row = A[user,:].toarray().squeeze()
            cut += row[indices].sum()
            vol += row.sum()
        if vol == 0:
            vol = 1
        cut = cut/vol
        norm_cut += cut
    ###########################################################
    return norm_cut
```

Notice, how using the unnormalized Laplacian leads to a much better ratio cut, while the normalized Laplacian leads to better normalized cut.

In [14]:
```
num_clusters = 6
```

In [15]:
```
np.random.seed(12903)
norm_laplacian = False
z_unnorm = spectral_clustering(A, num_clusters, norm_laplacian)
print('When using L_unnorm:')
print(' ratio cut = {:.3f}'.format(compute_ratio_cut(A, z_unnorm)))
print(' normalized cut = {:.3f}'.format(compute_normalized_cut(A, z_unnor
m)))
print(' sizes of partitions are: {}'.format([len(clust) for clust in labels_
to_list_of_clusters(z_unnorm)]))
```

```
When using L_unnorm:
 ratio cut = 369.109
 normalized cut = 5.000
 sizes of partitions are: [3379, 1, 1, 1, 1, 1]
```

In [16]:
```
np.random.seed(12323)
norm_laplacian = True
z_norm = spectral_clustering(A, num_clusters, norm_laplacian)
print('When using L_norm:')
print(' ratio cut = {:.3f}'.format(compute_ratio_cut(A, z_norm)))
print(' normalized cut = {:.3f}'.format(compute_normalized_cut(A, z_norm)))
print(' sizes of partitions are: {}'.format([len(clust) for clust in labels_
to_list_of_clusters(z_norm)]))
```

```
When using L_norm:
 ratio cut = 5942.851
 normalized cut = 4.343
 sizes of partitions are: [350, 742, 389, 754, 572, 577]
```

3. Visualize the results (5 pt)

In the final part of the assignment, your task is to print out the 5 most popular types of restaurants visited by the users in each cluster.

In [17]:
```
def print_top_categories_for_each_cluster(top_k: int, z: np.array,
                                          F: sp.csr_matrix,
                                          categories: List[str]):
    """Print the top-K categories among users in each cluster.
    For each cluster, the function prints names of the top-K categories,
    and number of users that like the respective category (separated by a co
mma).
    The function doesn't return anything, just prints the output.

    Parameters
    ----------
    top_k : int
        Number of most popular categories to print for each cluster.
    z : np.array, shape [N]
        Cluster labels.
    F : sp.csr_matrix, shape [N, C]
        Matrix that tells preferences of each user to each category.
        F[i, c] = 1 if user i gave at least one positive review to at least
one restaurant in category c.
    categories : list, shape [C]
        Names of the categories.

    """
    ############################################################
    # YOUR CODE HERE

    #num_clusters = 6 range(6)  0 1 2 3 4 5
    C = len(categories)
    N = len(z)
    for i in range(6):
        print('Most popular categories in cluster', i)
        score_categoties = np.zeros(C)
        for n in range(N):
            if z[n] == i:
                score_categoties += F[n, :]
        top_k_index = np.flip(np.argsort(score_categoties))[:top_k]
        for index in top_k_index:
            print('- ', categories[index], ',', int(score_categoties[inde
x]))

    ############################################################
```

In [18]:
```
np.random.seed(23142)
z_norm = spectral_clustering(A, num_clusters, True)
r = print_top_categories_for_each_cluster(5, z_norm, F, categories)
```

```
Most popular categories in cluster 0
-  Seafood , 315
-  Mexican , 314
-  Sandwiches , 294
-  Japanese , 291
-  Breakfast & Brunch , 286
Most popular categories in cluster 1
-  Breakfast & Brunch , 636
-  Sandwiches , 528
-  Italian , 514
-  Pizza , 482
-  Coffee & Tea , 473
Most popular categories in cluster 2
-  Specialty Food , 356
-  Thai , 355
-  Breakfast & Brunch , 348
-  Japanese , 333
-  Ethnic Food , 330
Most popular categories in cluster 3
-  Breakfast & Brunch , 664
-  Italian , 626
-  American (Traditional) , 518
-  Sandwiches , 518
-  Pizza , 485
Most popular categories in cluster 4
-  Japanese , 507
-  Breakfast & Brunch , 462
-  Sandwiches , 435
-  Italian , 417
-  Asian Fusion , 414
Most popular categories in cluster 5
-  Japanese , 529
-  Chinese , 441
-  Asian Fusion , 414
-  Sushi Bars , 408
-  Desserts , 406
```

In [ ]: