Modeling text data with a hierarchy of clusters

Hierarchical clustering refers to a class of clustering methods that seek to build a **hierarchy** of clusters, in which some clusters contain others. In this assignment, we will explore a top-down approach, recursively bipartitioning the data using k-means.

If you are using GraphLab Create

An IPython Notebook has been provided below to you for this assignment. This notebook contains the instructions, quiz questions and partially-completed code for you to use as well as some cells to test your code.

- Download the Wikipedia people dataset in SFrame format: people_wiki.gl.zip
- Download the companion IPython notebook:6_hierarchical_clustering_blank.ipynb
- Download a collection of helper functions:em_utilities.py
- Save the files in the same directory (where you are calling IPython notebook from) and unzip the data file and model file.

Open the companion IPython notebook and follow the instructions in the notebook. The instructions below do not apply to users of GraphLab Create.

If you are not using GraphLab Create

It is possible to complete this assignment without using GraphLab Create. The instructions below are geared towards Python users, but you are free to adapt them to your specific environment.

Disclaimer. We have tested the assessment using the standard Python installation (with access to scikit-learn). However, the assessment may not be compatible with other tools (e.g. Matlab, R).

Download the dataset

- Download the Wikipedia people dataset in SFrame format: people_wiki.gl.zip. (Those experimenting with other tools, get people_wiki.csv.zip instead.)
- Download the mapping between words and integer indices:people_wiki_map_index_to_word.gl.zip (or alternatively,people_wiki_map_index_to_word.json.zip)
- Download the pre-processed set of TF-IDF scores:people_wiki_tf_idf.npz

Import packages

```
import sframe                                    # see below for install
  instruction
import matplotlib.pyplot as plt
import numpy as np
```

```
from scipy.sparse import csr_matrix
from sklearn.cluster import KMeans                # we'll be using scikit
  -learn's KMeans for this assignment
from sklearn.metrics import pairwise_distances
from sklearn.preprocessing import normalize
%matplotlib inline
```

About SFrame. SFrame is a dataframe library Dato has released free-of-charge. Its source code is available here. You may install SFrame via pip:

```
pip install --upgrade sframe
```

Load in the dataset

```
wiki = sframe.SFrame('people_wiki.gl/')
```

As in the previous assignment, we extract the TF-IDF vector of each document.

For your convenience, we extracted the TF-IDF vectors from the dataset. The vectors are packaged in a sparse matrix, where the i-th row gives the TF-IDF vectors for the i-th document. Each column corresponds to a unique word appearing in the dataset.

To load in the TF-IDF vectors, run

```
def load_sparse_csr(filename):
    loader = np.load(filename)
    data = loader['data']
    indices = loader['indices']
    indptr = loader['indptr']
    shape = loader['shape']

    return csr_matrix( (data, indices, indptr), shape)

tf_idf = load_sparse_csr('people_wiki_tf_idf.npz')
map_index_to_word = sframe.SFrame('people_wiki_map_index_to_word.gl/')
```

To be consistent with the k-means assignment, let's normalize all vectors to have unit norm.

```
tf_idf = normalize(tf_idf)
```

(Optional) Extracting TF-IDF vectors yourself. We provide the pre-computed TF-IDF vectors to minimize potential compatibility issues. You are free to experiment with other tools to compute the TF-IDF vectors yourself. A good place to start issklearn.TfidfVectorizer. Note. Due to variations intokenization and other factors, your TF-IDF vectors may differ from the ones we provide. For the purpose the assessment, we ask you to use the vectors from people_wiki_tf_idf.npz.

Bipartition the Wikipedia dataset using k-means

Recall our workflow for clustering text data with k-means:

1. Load the dataframe containing a dataset, such as the Wikipedia text dataset.
2. Extract the data matrix from the dataframe.
3. Run k-means on the data matrix with some value of k.
4. Visualize the clustering results using the centroids, cluster assignments, and the original dataframe. We keep the original dataframe around because the data matrix does not keep auxiliary information (in the case of the text dataset, the title of each article).

Let us modify the workflow to perform bipartitioning:

1. Load the dataframe containing a dataset, such as the Wikipedia text dataset.
2. Extract the data matrix from the dataframe.
3. Run k-means on the data matrix with k=2.
4. Divide the data matrix into two parts using the cluster assignments.
5. Divide the dataframe into two parts, again using the cluster assignments. This step is necessary to allow for visualization.
6. Visualize the bipartition of data.

We'd like to be able to repeat Steps 3-6 multiple times to produce a **hierarchy** of clusters such as the following:

```
                   (root)
                      |
         +------------+-------------+
         |                          |
      Cluster                    Cluster
  +------+-----+             +------+-----+
  |            |             |            |
Cluster     Cluster       Cluster      Cluster
```

Each **parent cluster** is bipartitioned to produce two**child clusters**. At the very top is the **root cluster**, which consists of the entire dataset.

Now we write a wrapper function to bipartition a given cluster using k-means. There are three variables that together comprise the cluster:

- dataframe: a subset of the original dataframe that correspond to member rows of the cluster
- matrix: same set of rows, stored in sparse matrix format
- centroid: the centroid of the cluster (not applicable for the root cluster)

Rather than passing around the three variables separately, we package them into a Python dictionary. The wrapper function takes a single dictionary (representing a parent cluster) and returns two dictionaries (representing the child clusters).

```
def bipartition(cluster, maxiter=400, num_runs=4, seed=None):
    '''cluster: should be a dictionary containing the following keys
                * dataframe: original dataframe
                * matrix:    same data, in matrix format
                * centroid:  centroid for this particular cluster'''

    data_matrix = cluster['matrix']
    dataframe   = cluster['dataframe']

    # Run k-means on the data matrix with k=2. We use scikit-learn here to
      simplify workflow.
    kmeans_model = KMeans(n_clusters=2, max_iter=maxiter, n_init=num_runs,
      random_state=seed, n_jobs=-1)
    kmeans_model.fit(data_matrix)
    centroids, cluster_assignment = kmeans_model.cluster_centers_, kmeans_model
      .labels_

    # Divide the data matrix into two parts using the cluster assignments.
    data_matrix_left_child, data_matrix_right_child =
      data_matrix[cluster_assignment==0], \

                                          data_matrix[cluster_assignment==1]

    # Divide the dataframe into two parts, again using the cluster assignments.
    cluster_assignment_sa = sframe.SArray(cluster_assignment) # minor format
      conversion
    dataframe_left_child, dataframe_right_child     =
      dataframe[cluster_assignment_sa==0], \

                                          dataframe[cluster_assignment_sa==1]


    # Package relevant variables for the child clusters
    cluster_left_child  = {'matrix': data_matrix_left_child,
                           'dataframe': dataframe_left_child,
                           'centroid': centroids[0]}
    cluster_right_child = {'matrix': data_matrix_right_child,
                           'dataframe': dataframe_right_child,
                           'centroid': centroids[1]}

```

The following cell performs bipartitioning of the Wikipedia dataset. Allow 20-60 seconds to finish.

Note. For the purpose of the assignment, we set an explicit seed (seed=1) to produce identical outputs for every run. In pratical applications, you might want to use different random seeds for all runs.

```
wiki_data = {'matrix': tf_idf, 'dataframe': wiki} # no 'centroid' for the root
  cluster
left_child, right_child = bipartition(wiki_data, maxiter=100, num_runs=8, seed=1
  )
```

Let's examine the contents of one of the two clusters, which we call the left_child, referring to the tree visualization above.

```
print left_child
print right_child
```

Visualize the bipartition

We provide you with a modified version of the visualization function from the k-means assignment. For each cluster, we print the top 5 words with highest TF-IDF weights in the centroid and display excerpts for the 8 nearest neighbors of the centroid.

```
def display_single_tf_idf_cluster(cluster, map_index_to_word):
    '''map_index_to_word: SFrame specifying the mapping betweeen words and
      column indices'''

    wiki_subset   = cluster['dataframe']
    tf_idf_subset = cluster['matrix']
    centroid      = cluster['centroid']

    # Print top 5 words with largest TF-IDF weights in the cluster
    idx = centroid.argsort()[::-1]
    for i in xrange(5):
        print('{0:s}:{1:.3f}'.format(map_index_to_word['category'][idx[i]],
          centroid[idx[i]])),
    print('')

    # Compute distances from the centroid to all data points in the cluster.
    distances = pairwise_distances(tf_idf_subset, [centroid], metric='euclidean'
      ).flatten()
    # compute nearest neighbors of the centroid within the cluster.
    nearest_neighbors = distances.argsort()
    # For 8 nearest neighbors, print the title as well as first 180 characters
      of text.
    # Wrap the text at 80-character mark.
    for i in xrange(8):
        text = ' '.join(wiki_subset[nearest_neighbors[i]]['text'].split(None, 25
          )[0:25])
        print('* {0:50s} {1:.5f}\n  {2:s}\n  {3:s}'.format
          (wiki_subset[nearest_neighbors[i]]['name'],
            distances[nearest_neighbors[i]], text[:90], text[90:180] if len
              (text) > 90 else ''))
    print('')
```

Let's visualize the two child clusters:

```
display_single_tf_idf_cluster(left_child, map_index_to_word)
display_single_tf_idf_cluster(right_child, map_index_to_word)
```

The left cluster consists of athletes, whereas the right cluster consists of non-athletes. So far, we have a single-level hierarchy consisting of two clusters, as follows:

```
                                    Wikipedia
                                        +
                                        |
                   +--------------------------+--------------------+
                   |                                               |
                   +                                               +
                Athletes                                     Non-athletes
```

Is this hierarchy good enough? **When building a hierarchy of clusters, we must keep our particular application in mind.** For instance, we might want to build a **directory** for Wikipedia articles. A good directory would let you quickly narrow down your search to a small set of related articles. The categories of athletes and non-athletes are too general to facilitate efficient search. For this reason, we decide to build another level into our hierarchy of clusters with the goal of getting more specific cluster structure at the lower level. To that end, we subdivide both the athletes and non-athletes clusters.

Perform recursive bipartitioning

Cluster of athletes. To help identify the clusters we've built so far, let's give them easy-to-read aliases:

```
athletes = left_child
non_athletes = right_child
```

Using the bipartition function, we produce two child clusters of the athlete cluster:

```
# Bipartition the cluster of athletes
left_child_athletes, right_child_athletes = bipartition(athletes, maxiter=100,
  num_runs=8, seed=1)
```

The left child cluster mainly consists of baseball players. On the other hand, the right child cluster is a mix of football players and ice hockey players.

```
display_single_tf_idf_cluster(left_child_athletes, map_index_to_word)
display_single_tf_idf_cluster(right_child_athletes, map_index_to_word)
```

Note. Concerning use of "football"

The occurrences of the word "football" above refer to association football. This sports is also known as "soccer" in United States (to avoid confusion withAmerican football). We will use "football" throughout when discussing topic representation.

Our hierarchy of clusters now looks like this:

```
                                            Wikipedia
                                                +
                                                |
                     +--------------------------+--------------------+
                     |                                               |
                     +                                               +
                  Athletes                                      Non-athletes
                     +
                     |
         +-----------+--------+
         |                    |
         |                    +
         +                 football/
      baseball            ice hockey
```

Should we keep subdividing the clusters? If so, which cluster should we subdivide? To answer this question, we again think about our application. Since we organize our directory by topics, it would be nice to have topics that are about as coarse as each other. For instance, if one cluster is about baseball, we expect some other clusters about football, basketball, volleyball, and so forth. That is, **we would like to achieve similar level of granularity for all**

clusters.

Notice that the right child cluster is more coarse than the left child cluster. The right cluster possesses a greater variety of topics than the left (ice hockey/football vs. baseball). So the right child cluster should be subdivided further to produce finer child clusters.

Let's give the clusters aliases as well:

```
baseball            = left_child_athletes
ice_hockey_football = right_child_athletes
```

Cluster of ice hockey players and football players. In answering the following quiz question, take a look at the topics represented in the top documents (those closest to the centroid), as well as the list of words with highest TF-IDF weights.

Quiz Question. Bipartition the cluster of ice hockey and football players. Which of the two child clusters should be futher subdivided?

Note. To achieve consistent results, use the arguments maxiter=100, num_runs=8, seed=1 when calling the bipartition function.

1. The left child cluster
2. The right child cluster

Caution. The granularity criteria is an imperfect heuristic and must be taken with a grain of salt. It takes a lot of manual intervention to obtain a good hierarchy of clusters.

- **If a cluster is highly mixed, the top articles and words may not convey the full picture of the cluster.** Thus, we may be misled if we judge the purity of clusters solely by their top documents and words.
- **Many interesting topics are hidden somewhere inside the clusters but do not appear in the visualization.** We may need to subdivide further to discover new topics. For instance, subdividing the ice_hockey_football cluster led to the appearance of golf.

Quiz Question. Which diagram best describes the hierarchy right after splitting the**ice_hockey_football** cluster? Refer to the quiz form for the diagrams.

Cluster of non-athletes. Now let us subdivide the cluster of non-athletes.

```
# Bipartition the cluster of non-athletes
left_child_non_athletes, right_child_non_athletes = bipartition(non_athletes,
  maxiter=100, num_runs=8, seed=1)

display_single_tf_idf_cluster(left_child_non_athletes, map_index_to_word)
display_single_tf_idf_cluster(right_child_non_athletes, map_index_to_word)
```

The first cluster consists of scholars, politicians, and government officials whereas the second consists of musicians, artists, and actors. Run the following code cell to make convenient aliases for the clusters.

```
scholars_politicians_etc = left_child_non_athletes
musicians_artists_etc = right_child_non_athletes
```

Quiz Question. Let us bipartition the clusters**scholars_politicians_etc** and**musicians_artists_etc**. Which diagram best describes the resulting hierarchy of clusters for the non-athletes? Refer to the quiz for the diagrams.

Note. Use maxiter=100, num_runs=8, seed=1 for consistency of output.