Clustering text data with k-means

In this assignment you will

- Cluster Wikipedia documents using K-means
- Explore the role of random initialization on the quality of the clustering
- Explore how results differ after changing the number of clusters
- Evaluate clustering, both quantitatively and qualitatively

When properly executed, clustering uncovers valuable insights from a set of unlabeled documents.

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: 2_kmeans-with-text-data_blank.ipynb
- Download the NumPy array containing pre-computed list of clusters: kmeans-arrays.npz
- Save all the files in the same directory (where you are calling IPython notebook from) and unzip the data 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
- Download the NumPy array containing pre-computed list of clusters: kmeans-arrays.npz

Import packages

```
import sframe                                              # see below for install
    instruction
import matplotlib.pyplot as plt                            # plotting
import numpy as np                                         # dense matrices
from scipy.sparse import csr_matrix                        # sparse matrices
from sklearn.preprocessing import normalize                # normalizing vectors
from sklearn.metrics import pairwise_distances             # pairwise distances
import sys
import os
%matplotlib inline
```

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

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

Load data, extract features

To work with text data, we must first convert the documents into numerical features. As in the first assignment, let's extract TF-IDF features for each article.

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.

```
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)

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

(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 is sklearn.TfidfVectorizer. Note. Due to variations in tokenization 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.

Normalize all vectors

As discussed in the previous assignment, Euclidean distance can be a poor metric of similarity between documents, as it unfairly penalizes long articles. For a reasonable assessment of similarity, we should disregard the length information and use length-agnostic metrics, such as cosine distance.

The k-means algorithm does not directly work with cosine distance, so we take an alternative route to remove length information: we normalize all vectors to be unit length. It turns out that Euclidean distance closely mimics cosine distance when all vectors are unit length. In particular, the squared Euclidean distance between any two vectors of length one is directly proportional to their cosine distance.

We can prove this as follows. Let $\mathbf{x}$ and $\mathbf{y}$ be normalized vectors, i.e. unit vectors, so that $||\mathbf{x}|| = ||\mathbf{y}|| = 1$. Write the squared Euclidean distance as the dot product of $(\mathbf{x} - \mathbf{y})$ to itself:

$$||\mathbf{x} - \mathbf{y}||^2 = (\mathbf{x} - \mathbf{y})^T(\mathbf{x} - \mathbf{y}) = ||\mathbf{x}||^2 - 2(\mathbf{x}^T\mathbf{y}) + ||\mathbf{y}||^2 = 2 - 2(\mathbf{x}^T\mathbf{y}) = 2\left(1 - \frac{\mathbf{x}^T\mathbf{y}}{||\mathbf{x}||\,||\mathbf{y}||}\right)$$

This tells us that two **unit vectors** that are close in Euclidean distance are also close in cosine distance. Thus, the k-means algorithm (which naturally uses Euclidean distances) on normalized vectors will produce the same results as clustering using cosine distance as a distance metric.

We import the normalize() function from scikit-learn to normalize all vectors to unit length.

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

Implement k-means

Let us implement the k-means algorithm. First, we choose an initial set of centroids. A common practice is to choose randomly from the data points.

Note: We specify a seed here, so that everyone gets the same answer. In practice, we highly recommend to use different seeds every time (for instance, by using the current timestamp).

```python
def get_initial_centroids(data, k, seed=None):
    '''Randomly choose k data points as initial centroids'''
    if seed is not None: # useful for obtaining consistent results
        np.random.seed(seed)
    n = data.shape[0] # number of data points

    # Pick K indices from range [0, N).
    rand_indices = np.random.randint(0, n, k)

    # Keep centroids as dense format, as many entries will be nonzero due to averaging.
    # As long as at least one document in a cluster contains a word,
    # it will carry a nonzero weight in the TF-IDF vector of the centroid.
    centroids = data[rand_indices,:].toarray()

    return centroids
```

After initialization, the k-means algorithm iterates between the following two steps:

1. Assign each data point to the closest centroid.
2. Revise centroids as the mean of the assigned data points.

In pseudocode, we iteratively do the following:

```
cluster_assignment = assign_clusters(data, centroids)
centroids = revise_centroids(data, k, cluster_assignment)
```

Assigning clusters. How do we implement Step 1 of the main k-means loop above? First import pairwise_distances function from scikit-learn, which calculates Euclidean distances between rows of given arrays. See this documentation for more information.

For the sake of demonstration, let's look at documents 100 through 102 as query documents and compute the distances between each of these documents and every other document in the corpus. In the k-means algorithm, we will have to compute pairwise distances between the set of centroids and the set of documents.

```
# Get the TF-IDF vectors for documents 100 through 102.
queries = tf_idf[100:102,:]

# Compute pairwise distances from every data point to each query vector.
dist = pairwise_distances(tf_idf, queries, metric='euclidean')

print dist
```

This cell outputs a 2D array of form

```
[[ 1.41000789  1.36894636]
 [ 1.40935215  1.41023886]
 [ 1.39855967  1.40890299]
 ...,
 [ 1.41108296  1.39123646]
 [ 1.41022804  1.31468652]
 [ 1.39899784  1.41072448]]
```

More formally, dist[i,j] is assigned the distance between the ith row of X (i.e., X[i,:]) and the jth row of Y (i.e., Y[j,:]).

Checkpoint: For a moment, suppose that we initialize three centroids with the first 3 rows of tf_idf. Write code to compute distances from each of the centroids to all data points in tf_idf. Then find the distance between row 430 of tf_idf and the second centroid and save it to **dist**. Run the following cell to check your answer.

```
'''Test cell'''
if np.allclose(dist, pairwise_distances(tf_idf[430,:], tf_idf[1,:])):
    print('Pass')
else:
    print('Check your code again')
```

Checkpoint: Next, given the pairwise distances, we take the minimum of the distances for each data point. Fittingly, NumPy provides an argmin function. See this documentation for details.

Read the documentation and write code to produce a 1D array whose i-th entry indicates the centroid that is the closest to the i-th data point. Use the list of distances from the previous checkpoint and save them as distances. The value 0 indicates closeness to the first centroid, 1 indicates closeness to the second centroid, and so forth. Save this array as **closest_cluster**. Run the following cell to check your answer.

Hint: the resulting array should be as long as the number of data points.

```python
'''Test cell'''
reference = [list(row).index(min(row)) for row in distances]
if np.allclose(closest_cluster, reference):
    print('Pass')
else:
    print('Check your code again')
```

Checkpoint: Let's put these steps together. First, initialize three centroids with the first 3 rows of tf_idf. Then, compute distances from each of the centroids to all data points in tf_idf. Finally, use these distance calculations to compute cluster assignments and assign them to cluster_assignment. Run the following cell to check your code.

```python
if len(cluster_assignment)==59071 and \
   np.array_equal(np.bincount(cluster_assignment), np.array([23061, 10086, 25924])):
    print('Pass') # count number of data points for each cluster
else:
    print('Check your code again.')
```

Now we are ready to fill in the blanks in this function:

```python
def assign_clusters(data, centroids):

    # Compute distances between each data point and the set of centroids:
    # Fill in the blank (RHS only)
    distances_from_centroids = ...

    # Compute cluster assignments for each data point:
    # Fill in the blank (RHS only)
    cluster_assignment = ...

    return cluster_assignment
```

which is simply generalization of what we did above.

Checkpoint. For the last time, let us check if Step 1 was implemented correctly. With rows 0, 2, 4, and 6 of tf_idf as an initial set of centroids, we assign cluster labels to rows 0, 10, 20, ..., and 90 of tf_idf. The resulting cluster labels should be [0, 1, 1, 0, 0, 2, 0, 2, 2, 1].

```python
if np.allclose(assign_clusters(tf_idf[0:100:10], tf_idf[0:8:2]), np.array([0, 1, 1, 0, 0, 2, 0, 2
    , 2, 1])):
    print('Pass')
else:
    print('Check your code again.')
```

Revising clusters

Let's turn to Step 2, where we compute the new centroids given the cluster assignments.

SciPy and NumPy arrays allow for filtering via Boolean masks. For instance, we filter all data points that are assigned to cluster 0 by writing "data[cluster_assignment==0,:]".

To develop intuition about filtering, let's look at a toy example consisting of 3 data points and 2 clusters.

```
data = np.array([[1., 2., 0.],
                 [0., 0., 0.],
                 [2., 2., 0.]])
centroids = np.array([[0.5, 0.5, 0.],
                      [0., -0.5, 0.]])
```

Let's assign these data points to the closest centroid.

```
cluster_assignment = assign_clusters(data, centroids)
print cluster_assignment   # prints [0 1 0]
```

The expression "cluster_assignment==1" gives a list of Booleans that says whether each data point is assigned to cluster 1 or not. For cluster 0, the expression is "cluster_assignment==0". In lieu of indices, we can put in the list of Booleans to pick and choose rows. Only the rows that correspond to a True entry will be retained.

First, let's look at the data points (i.e., their values) assigned to cluster 1:

```
print data[cluster_assignment==1]
```

The output makes sense since [0 0 0] is closer to [0 -0.5 0] than to [0.5 0.5 0].

Now let's look at the data points assigned to cluster 0:

```
print data[cluster_assignment==0]
```

Again, this makes sense since these values are each closer to [0.5 0.5 0] than to [0 -0.5 0].

Given all the data points in a cluster, it only remains to compute the mean. Use np.mean(). By default, the function averages all elements in a 2D array. To compute row-wise or column-wise means, add the axis argument. See the linked documentation for details.

Use this function to average the data points in cluster 0:

```
print data[cluster_assignment==0].mean(axis=0)
```

We are now ready to complete this function:

```
def revise_centroids(data, k, cluster_assignment):
    new_centroids = []
    for i in xrange(k):
        # Select all data points that belong to cluster i. Fill in the blank (RHS only)
        member_data_points = ...
        # Compute the mean of the data points. Fill in the blank (RHS only)
        centroid = ...

        # Convert numpy.matrix type to numpy.ndarray type
        centroid = centroid.A1
        new_centroids.append(centroid)
    new_centroids = np.array(new_centroids)

    return new_centroids
```

Checkpoint. Let's check our Step 2 implementation. Letting rows 0, 10, ..., 90 of tf_idf as the data points and the cluster labels [0, 1, 1, 0, 0, 2, 0, 2, 2, 1], we compute the next set of centroids. Each centroid is given by the average of all member data points in corresponding cluster.

```
result = revise_centroids(tf_idf[0:100:10], 3, np.array([0, 1, 1, 0, 0, 2, 0, 2, 2, 1]))
if np.allclose(result[0], np.mean(tf_idf[[0,30,40,60]].toarray(), axis=0)) and \
   np.allclose(result[1], np.mean(tf_idf[[10,20,90]].toarray(), axis=0))   and \
   np.allclose(result[2], np.mean(tf_idf[[50,70,80]].toarray(), axis=0)):
    print('Pass')
else:
    print('Check your code')
```

Assessing convergence

How can we tell if the k-means algorithm is converging? We can look at the cluster assignments and see if they stabilize over time. In fact, we'll be running the algorithm until the cluster assignments stop changing at all. To be extra safe, and to assess the clustering performance, we'll be looking at an additional criteria: the sum of all squared distances between data points and centroids. This is defined as

$$J(\mathcal{Z},\mu) = \sum_{j=1}^{k} \sum_{i:z_i=j} \|\mathbf{x}_i - \mu_j\|^2.$$

The smaller the distances, the more homogeneous the clusters are. In other words, we'd like to have "tight" clusters.

```python
def compute_heterogeneity(data, k, centroids, cluster_assignment):

    heterogeneity = 0.0
    for i in xrange(k):

        # Select all data points that belong to cluster i. Fill in the blank (RHS only)
        member_data_points = data[cluster_assignment==i, :]

        if member_data_points.shape[0] > 0: # check if i-th cluster is non-empty
            # Compute distances from centroid to data points (RHS only)
            distances = pairwise_distances(member_data_points, [centroids[i]], metric
                ='euclidean')
            squared_distances = distances**2
            heterogeneity += np.sum(squared_distances)

    return heterogeneity
```

Combining into a single function

Once the two k-means steps have been implemented, as well as our heterogeneity metric we wish to monitor, it is only a matter of putting these functions together to write a k-means algorithm that

- Repeatedly performs Steps 1 and 2
- Tracks convergence metrics
- Stops if either no assignment changed or we reach a certain number of iterations.

```python
# Fill in the blanks
def kmeans(data, k, initial_centroids, maxiter, record_heterogeneity=None, verbose=False):
    '''This function runs k-means on given data and initial set of centroids.
       maxiter: maximum number of iterations to run.
       record_heterogeneity: (optional) a list, to store the history of heterogeneity as function of iterations
                             if None, do not store the history.
       verbose: if True, print how many data points changed their cluster labels in each iteration'''
    centroids = initial_centroids[:]
    prev_cluster_assignment = None

    for itr in xrange(maxiter):
        if verbose:
            print(itr)

        # 1. Make cluster assignments using nearest centroids
        # YOUR CODE HERE
        cluster_assignment = ...

        # 2. Compute a new centroid for each of the k clusters, averaging all data points assigned to that cluster.
        # YOUR CODE HERE
        centroids = ...

        # Check for convergence: if none of the assignments changed, stop
        if prev_cluster_assignment is not None and \
          (prev_cluster_assignment==cluster_assignment).all():
            break

        # Print number of new assignments
        if prev_cluster_assignment is not None:
            num_changed = sum(abs(prev_cluster_assignment-cluster_assignment))
            if verbose:
                print('    {0:5d} elements changed their cluster assignment.'.format(num_changed))

        # Record heterogeneity convergence metric
        if record_heterogeneity is not None:
            # YOUR CODE HERE
            score = ...
            record_heterogeneity.append(score)

        prev_cluster_assignment = cluster_assignment[:]

    return centroids, cluster_assignment
```

Plotting convergence metric

We can use the above function to plot the convergence metric across iterations.

```python
def plot_heterogeneity(heterogeneity, k):
    plt.figure(figsize=(7,4))
    plt.plot(heterogeneity, linewidth=4)
    plt.xlabel('# Iterations')
    plt.ylabel('Heterogeneity')
    plt.title('Heterogeneity of clustering over time, K={0:d}'.format(k))
    plt.rcParams.update({'font.size': 16})
    plt.tight_layout()
```

Let's consider running k-means with K=3 clusters for a maximum of 400 iterations, recording cluster heterogeneity at every step. Then, let's plot the heterogeneity over iterations using the plotting function above.

```python
k = 3
heterogeneity = []
initial_centroids = get_initial_centroids(tf_idf, k, seed=0)
centroids, cluster_assignment = kmeans(tf_idf, k, initial_centroids, maxiter=400,
                                       record_heterogeneity=heterogeneity, verbose=True)
plot_heterogeneity(heterogeneity, k)
```

Quiz Question. (True/False) The clustering objective (heterogeneity) is non-increasing for this example.

Quiz Question. Let's step back from this particular example. If the clustering objective (heterogeneity) would ever increase when running k-means, that would indicate: (choose one)

1. k-means algorithm got stuck in a bad local minimum
2. There is a bug in the k-means code
3. All data points consist of exact duplicates
4. Nothing is wrong. The objective should generally go down sooner or later.

Quiz Question. Which of the cluster contains the greatest number of data points in the end? Hint: Use np.bincount() to count occurrences of each cluster label.

1. Cluster #0
2. Cluster #1
3. Cluster #2

Beware of local minima

One weakness of k-means is that it tends to get stuck in a local minimum. To see this, let us run k-means multiple times, with different initial centroids created using different random seeds.

Note: Again, in practice, you should set different seeds for every run. We give you a list of seeds for this assignment so that everyone gets the same answer.

This may take several minutes to run.

```python
k = 10
heterogeneity = {}
import time
start = time.time()
for seed in [0, 20000, 40000, 60000, 80000, 100000, 120000]:
    initial_centroids = get_initial_centroids(tf_idf, k, seed)
    centroids, cluster_assignment = kmeans(tf_idf, k, initial_centroids, maxiter=400,
                                           record_heterogeneity=None, verbose=False)
    # To save time, compute heterogeneity only once in the end
    heterogeneity[seed] = compute_heterogeneity(tf_idf, k, centroids, cluster_assignment)
    print('seed={0:06d}, heterogeneity={1:.5f}'.format(seed, heterogeneity[seed]))
    sys.stdout.flush()
end = time.time()
print(end-start)
```

Notice the variation in heterogeneity for different initializations. This indicates that k-means sometimes gets stuck at a bad local minimum.

Quiz Question. Another way to capture the effect of changing initialization is to look at the distribution of cluster assignments. Add a line to the code above to compute the size (# of member data points) of clusters for each run of k-means. Look at the size of the largest cluster (most # of member data points) across multiple runs, with seeds 0, 20000, ..., 120000. How much does this measure vary across the runs? What is the minimum and maximum values this quantity takes?

One effective way to counter this tendency is to use **k-means++** to provide a smart initialization. This method tries to spread out the initial set of centroids so that they are not too close together. It is known to improve the quality of local optima and lower average runtime.

```python
def smart_initialize(data, k, seed=None):
    '''Use k-means++ to initialize a good set of centroids'''
    if seed is not None: # useful for obtaining consistent results
        np.random.seed(seed)
    centroids = np.zeros((k, data.shape[1]))

    # Randomly choose the first centroid.
    # Since we have no prior knowledge, choose uniformly at random
    idx = np.random.randint(data.shape[0])
    centroids[0] = data[idx,:].toarray()
    # Compute distances from the first centroid chosen to all the other data points
    distances = pairwise_distances(data, centroids[0:1], metric='euclidean').flatten()

    for i in xrange(1, k):
        # Choose the next centroid randomly, so that the probability for each data point to be chosen
        # is directly proportional to its squared distance from the nearest centroid.
        # Roughtly speaking, a new centroid should be as far as from ohter centroids as possible.
        idx = np.random.choice(data.shape[0], 1, p=distances/sum(distances))
        centroids[i] = data[idx,:].toarray()
        # Now compute distances from the centroids to all data points
        distances = np.min(pairwise_distances(data, centroids[0:i+1], metric='euclidean'),axis=1)

    return centroids
```

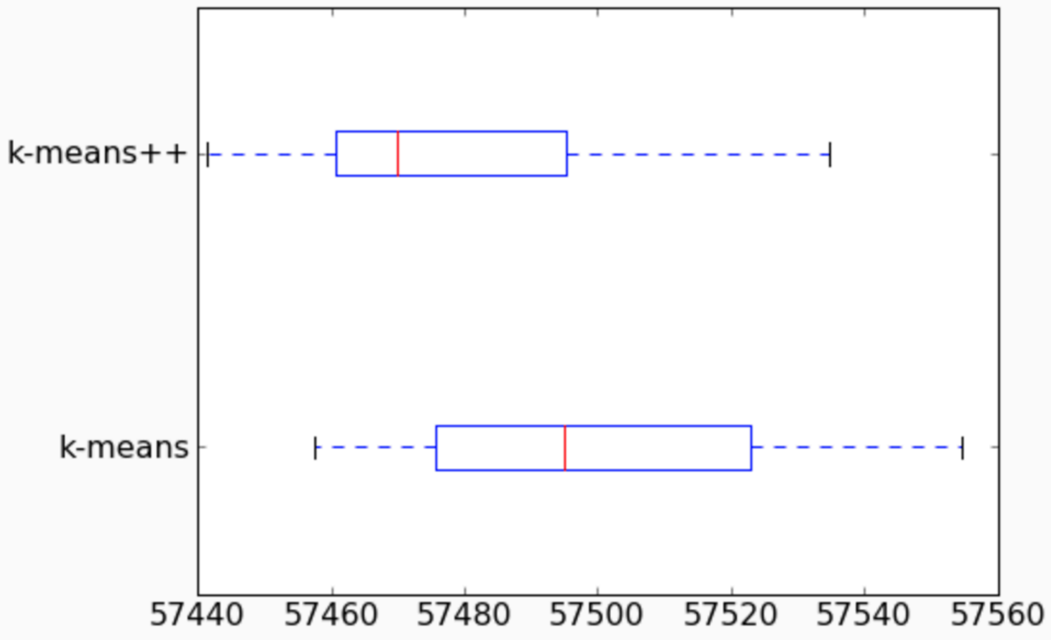
Let's now rerun k-means with 10 clusters using the same set of seeds, but always using k-means++ to initialize the algorithm.

This may take several minutes to run.

```python
k = 10
heterogeneity_smart = {}
start = time.time()
for seed in [0, 20000, 40000, 60000, 80000, 100000, 120000]:
    initial_centroids = smart_initialize(tf_idf, k, seed)
    centroids, cluster_assignment = kmeans(tf_idf, k, initial_centroids, maxiter=400,
                                           record_heterogeneity=None, verbose=False)
    # To save time, compute heterogeneity only once in the end
    heterogeneity_smart[seed] = compute_heterogeneity(tf_idf, k, centroids, cluster_assignment)
    print('seed={0:06d}, heterogeneity={1:.5f}'.format(seed, heterogeneity_smart[seed]))
    sys.stdout.flush()
end = time.time()
print(end-start)
```

Let's compare the set of cluster heterogeneities we got from our 7 restarts of k-means using random initialization compared to the 7 restarts of k-means using k-means++ as a smart initialization. The following code produces a box plot for each of these methods, indicating the spread of values produced by each method.

```python
plt.figure(figsize=(8,5))
plt.boxplot([heterogeneity.values(), heterogeneity_smart.values()], vert=False)
plt.yticks([1, 2], ['k-means', 'k-means++'])
plt.rcParams.update({'font.size': 16})
plt.tight_layout()
```



A few things to notice from the box plot:

- Random initialization results in a worse clustering than k-means++ on average.
- The best result of k-means++ is better than the best result of random initialization.

In general, you should run k-means at least a few times with different initializations and then return the run resulting in the lowest heterogeneity. Let us write a function that runs k-means multiple times and picks the best run that minimizes heterogeneity. The function accepts an optional list of seed values to be used for the multiple runs; if no such list is provided, the current UTC time is used as seed values.

```python
def kmeans_multiple_runs(data, k, maxiter, num_runs, seed_list=None, verbose=False):
    heterogeneity = {}

    min_heterogeneity_achieved = float('inf')
    best_seed = None
    final_centroids = None
    final_cluster_assignment = None

    for i in xrange(num_runs):

        # Use UTC time if no seeds are provided
        if seed_list is not None:
            seed = seed_list[i]
            np.random.seed(seed)
        else:
            seed = int(time.time())
            np.random.seed(seed)

        # Use k-means++ initialization
        # YOUR CODE HERE
        initial_centroids = ...

        # Run k-means
        # YOUR CODE HERE
        centroids, cluster_assignment = ...

        # To save time, compute heterogeneity only once in the end
        # YOUR CODE HERE
        heterogeneity[seed] = ...

        if verbose:
            print('seed={0:06d}, heterogeneity={1:.5f}'.format(seed, heterogeneity[seed]))
            sys.stdout.flush()

        # if current measurement of heterogeneity is lower than previously seen,
        # update the minimum record of heterogeneity.
        if heterogeneity[seed] < min_heterogeneity_achieved:
            min_heterogeneity_achieved = heterogeneity[seed]
            best_seed = seed
            final_centroids = centroids
            final_cluster_assignment = cluster_assignment

    # Return the centroids and cluster assignments that minimize heterogeneity.
    return final_centroids, final_cluster_assignment
```

How to choose K

Since we are measuring the tightness of the clusters, a higher value of K reduces the possible heterogeneity metric by definition. For example, if we have N data points and set K=N clusters, then we could have 0 cluster heterogeneity by setting the N centroids equal to the values of the N data points. (Note: Not all runs for larger K will result in lower heterogeneity than a single run with smaller K due to local optima.) Let's see explore this general trend for ourselves by performing the following analysis.

Use the kmeans_multiple_runs function to run k-means with five different values of K. For each K, use k-means++ and **multiple run**s to pick the best solution. In what follows, we consider K=2,10,25,50,100 and 7 restarts for each setting.

IMPORTANT: The code block below will take about one hour to finish. We highly suggest that you use the arrays that we have computed for you. Side note: In practice, a good implementation of k-means would utilize parallelism to run multiple runs of k-means at once. For an example, see scikit-learn's KMeans.

```
#def plot_k_vs_heterogeneity(k_values, heterogeneity_values):
#    plt.figure(figsize=(7,4))
#    plt.plot(k_values, heterogeneity_values, linewidth=4)
#    plt.xlabel('K')
#    plt.ylabel('Heterogeneity')
#    plt.title('K vs. Heterogeneity')
#    plt.rcParams.update({'font.size': 16})
#    plt.tight_layout()

#start = time.time()
#centroids = {}
#cluster_assignment = {}
#heterogeneity_values = []
#k_list = [2, 10, 25, 50, 100]
#seed_list = [0, 20000, 40000, 60000, 80000, 100000, 120000]

#for k in k_list:
#    heterogeneity = []
#    centroids[k], cluster_assignment[k] = kmeans_multiple_runs(tf_idf, k, maxiter=400,
#                                                               num_runs=len(seed_list),
#                                                               seed_list=seed_list,
#                                                               verbose=True)
#    score = compute_heterogeneity(tf_idf, k, centroids[k], cluster_assignment[k])
#    heterogeneity_values.append(score)

#plot_k_vs_heterogeneity(k_list, heterogeneity_values)

#end = time.time()
#print(end-start)
```

To use the pre-computed NumPy arrays, first download kmeans-arrays.npz as mentioned in the reading for this assignment and load them with the following code. Make sure the downloaded file is in the same directory as this notebook.

```python
def plot_k_vs_heterogeneity(k_values, heterogeneity_values):
    plt.figure(figsize=(7,4))
    plt.plot(k_values, heterogeneity_values, linewidth=4)
    plt.xlabel('K')
    plt.ylabel('Heterogeneity')
    plt.title('K vs. Heterogeneity')
    plt.rcParams.update({'font.size': 16})
    plt.tight_layout()

filename = 'kmeans-arrays.npz'

heterogeneity_values = []
k_list = [2, 10, 25, 50, 100]

if os.path.exists(filename):
    arrays = np.load(filename)
    centroids = {}
    cluster_assignment = {}
    for k in k_list:
        print k
        sys.stdout.flush()
        centroids[k] = arrays['centroids_{0:d}'.format(k)]
        cluster_assignment[k] = arrays['cluster_assignment_{0:d}'.format(k)]
        score = compute_heterogeneity(tf_idf, k, centroids[k], cluster_assignment[k])
        heterogeneity_values.append(score)

    plot_k_vs_heterogeneity(k_list, heterogeneity_values)

else:
    print('File not found. Skipping.')
```