In [78]:
```
%%javascript
/*****************************************************************************
Known Mathjax Issue with Chrome - a rounding issue adds a border to the righ
https://github.com/mathjax/MathJax/issues/1300
A quick hack to fix this based on stackoverflow discussions:
http://stackoverflow.com/questions/34277967/chrome-rendering-mathjax-equatic
*****************************************************************************

$('.math>span').css("border-left-color","transparent")
```

```
<IPython.core.display.Javascript object>
```

MIDS - w261 Machine Learning At Scale

Course Lead: Dr James G. Shanahan (**email** Jimi via James.Shanahan *AT* gmail.com)

Assignment - HW4

Name: *Chris Caudill*
Class: MIDS w261 (Section *Spring 2017 Group 1*)
Email: *cscaudill*@iSchool.Berkeley.edu
StudentId 3032134574 **End of StudentId**
Week: 4

NOTE: please replace `1234567` with your student id above
Due Time: HW is due the Tuesday of the following week by 8AM (West coast time). I.e., Tuesday, Feb 7, 2017 in the case of this homework.

Table of Contents

1 Instructions

Back to Table of Contents

MIDS UC Berkeley, Machine Learning at Scale DATSCIW261 ASSIGNMENT #4

Version 2017-26-1

IMPORTANT

This homework can be completed locally on your computer

=== INSTRUCTIONS for SUBMISSIONS ===

Follow the instructions for submissions carefully.

NEW: Going forward, each student will have a `HW-<user>` repository for all assignments.

Click this link to enable you to create a github repo within the MIDS261 Classroom: https://classroom.github.com/assignment-invitations/3b1d6c8e58351209f9dd865537111ff8 (https://classroom.github.com/assignment-invitations/3b1d6c8e58351209f9dd865537111ff8) and follow the instructions to create a HW repo.

Push the following to your HW github repo into the master branch:

- Your local HW4 directory. Your repo file structure should look like this:

```
HW-<user>
    --HW3
       |__MIDS-W261-HW-03-<Student_id>.ipnb
       |__MIDS-W261-HW-03-<Student_id>.pdf
       |__some other hw3 file
    --HW4
       |__MIDS-W261-HW-04-<Student_id>.ipnb
       |__MIDS-W261-HW-04-<Student_id>.pdf
       |__some other hw4 file
    etc..
```

2 Useful References

Back to Table of Contents

- See async lecture and live lecture

HW Problems

Back to Table of Contents

HW4.0

Back to Table of Contents

What is MrJob? How is it different to Hadoop MapReduce? What are the mapper_init, mapper_final(), combiner_final(), reducer_final() methods? When are they called?

MRJob is a python package used for running hadoop streaming jobs. It was developed by Yelp and assists in producing multistep jobs.

HW4.1

Back to Table of Contents What is serialization in the context of MrJob or Hadoop? When it used in these frameworks? What is the default serialization mode for input and outputs for MrJob?

MRJob expects data to be in bytes. To ensure the data is in byte-form, it uses input and output protocols to read byte-data in and convert it to Python objects, then afterwards to convert it back to byte-form.

The default input protocol is RawValueProtocol. The default output protocol is JSONProtocol.

In [1]:
```
%%bash
curl -L http://archive.ics.uci.edu/ml/machine-learning-databases/anonymous/a
```

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  C
urrent
                                 Dload  Upload   Total   Spent    Left  S
peed
100 1389k  100 1389k    0     0  1234k      0  0:00:01  0:00:01 --:--:--
 1423k
```

HW4.2 Preprocess log file data

Back to Table of Contents

Recall the Microsoft logfiles data from the async lecture. The logfiles are described are located at:

https://kdd.ics.uci.edu/databases/msweb/msweb.html (https://kdd.ics.uci.edu/databases/msweb/msweb.html) http://archive.ics.uci.edu/ml/machine-learning-databases/anonymous/ (http://archive.ics.uci.edu/ml/machine-learning-databases/anonymous/)

This dataset captures which areas (Vroots) of www.microsoft.com each user visited in a one-week timeframe in Feburary 1998.

Data Format

```
The data is in an ASCII-based sparse-data format called "DST". Each
 line of the data file starts with a letter which tells the line's t
ype. The three line types of interest are:
-- Attribute lines:
For example, 'A,1277,1,"NetShow for PowerPoint","/stream"'
Where:
  'A' marks this as an attribute line,
  '1277' is the attribute ID number for an area of the website (call
ed a Vroot),
  '1' may be ignored,
  '"NetShow for PowerPoint"' is the title of the Vroot,
  '"/stream"' is the URL relative to "http://www.microsoft.com (htt
p://www.microsoft.com)"

Case and Vote Lines:
For each user, there is a case line followed by zero or more vote li
nes.
For example:
  C,"10164",10164
  V,1123,1
  V,1009,1
  V,1052,1
Where:
  'C' marks this as a case line,
  '10164' is the case ID number of a user,
  'V' marks the vote lines for this case,
  '1123', 1009', 1052' are the attributes ID's of Vroots that a user
 visited.
  '1' may be ignored.
```

Here, you must transform/preprocess the data on a single node (i.e., not on a cluster of nodes) from the following format:

- C,"10001",10001 #Visitor id 10001
- V,1000,1 #Visit by Visitor 10001 to page id 1000
- V,1001,1 #Visit by Visitor 10001 to page id 1001
- V,1002,1 #Visit by Visitor 10001 to page id 1002
- C,"10002",10002 #Visitor id 10001
- V
- Note: #denotes comments

to the following format (V, PageID, 1, C, Visitor):

- V,1000,1,C, 10001
- V,1001,1,C, 10001
- V,1002,1,C, 10001

Write the python code to accomplish this transformation.

In [19]:
```
#START STUDENT CODE42
!rm ./processed_anonymous-msweb.data
with open('anonymous-msweb.data', 'r') as inputfile, open('processed_anonymo
    for line in inputfile:
        v = line.split(',')
        # If the line is customer information line, save it to customer_info
        if(v[0]=='C'):
            customer_info = v

        # If the line visit information line, concatenate the line with cust
        elif(v[0]=='V'):
            outputfile.write(line.strip()+','+customer_info[0]+','+customer_

        # If attribute line, print line
        elif(v[0]=='A'):
            outputfile.write(line)

#END STUDENT CODE42
```

In [20]:
```
!head -500 processed_anonymous-msweb.data
!wc -l processed_anonymous-msweb.data
```

HW4.3 Find the most frequent pages

Back to Table of Contents

Find the 5 most frequently visited pages using MrJob from the output of 4.2 (i.e., transfromed log file). You may get a weird result.

In [43]:
```
%%writefile MostFrequentVisits.py
#!/usr/bin/env python
#START STUDENT CODE43

from mrjob.job import MRJob
import csv

def csv_readline(line):
#     """Given a sting CSV line, return a list of strings."""
    for row in csv.reader([line]):
        return row

class SiteVisit(MRJob):

    def mapper(self, line_no, line):
#         """Extracts the site that was visited"""
        cell = csv_readline(line)
        if cell[0] == 'V':
            yield cell[1],1

    def reducer(self, site, visit_counts):
#         """Sumarizes the visit counts by adding them together."""
        total = sum(i for i in visit_counts)
        yield site, total

if __name__ == '__main__':
    SiteVisit.run()

#END STUDENT CODE43
```

```
Overwriting MostFrequentVisits.py
```

In [44]:
```
!chmod a+x MostFrequentVisits.py
```

In [45]:
```
%reload_ext autoreload
%autoreload 2
from MostFrequentVisits import SiteVisit
import csv
import operator
d={}

mr_job = SiteVisit(args=['processed_anonymous-msweb.data'])

with mr_job.make_runner() as runner:
    runner.run()
    for line in runner.stream_output():
        key, value = mr_job.parse_output_line(line)
        d[key] = value
sorted_d = sorted(d.items(), key=operator.itemgetter(1),reverse=True)
for key, value in sorted_d[0:5]:
    print '%s\t%s' % (key, value)
```

```
1008    10836
1034    9383
1004    8463
1018    5330
1017    5108
```

HW4.4 Find the most frequent visitor

Back to Table of Contents

Find the most frequent visitor of each page using MrJob and the output of 4.2 (i.e., transfromed log file). In this output please include the webpage URL, webpageID and Visitor ID. HINT: The maximum visits to any given webpage is 1.

In [88]:
```
%%writefile mostFrequentVisitors.py
#!/usr/bin/env python
#START STUDENT CODE44

from mrjob.job import MRJob
import csv

d={}

def csv_readline(line):
#     """Given a sting CSV line, return a list of strings."""
    for row in csv.reader([line]):
        return row

class FrequentVisitor(MRJob):

    def mapper(self, line_no, line):
#         """Extracts the site that was visited"""
        cell = csv_readline(line)
        if cell[0] == 'A':
            d[cell[1]] = cell[3]
        if cell[0] == 'V':
            key = (cell[1],cell[4], d[cell[1]])
            yield key,1

    def reducer(self, key, visit_counts):
#         """Sumarizes the visit counts by adding them together."""
        total = sum(i for i in visit_counts)
        yield key, total

if __name__ == '__main__':
    FrequentVisitor.run()

#END STUDENT CODE44
```

```
Overwriting mostFrequentVisitors.py
```

In [89]:
```
!chmod a+x mostFrequentVisitors.py
```

In [90]:
```
%reload_ext autoreload
%autoreload 2
from MostFrequentVisitors import FrequentVisitor
import csv
import operator

d={}
d_max={}
d_site={}
cur_max = 0

mr_job = FrequentVisitor(args=['processed_anonymous-msweb.data'])

with mr_job.make_runner() as runner:
    runner.run()
    for line in runner.stream_output():
        key, value = mr_job.parse_output_line(line)
        site = key[0]
        user = key[1]
        d_site[key[0]] = key[2]
        visits = value

        d[site] = (user, visits)

        for key in d:
            if key in d_max:
                if d[key] > d_max[key]:
                    d_max[key] = d[key]
            else:
                d_max[key] = d[key]
for k in d_max:
    print '%s\t%s\t%s' % (k,d_max[k][0], d_site[k])
```

HW4.5 Clustering Tweet Dataset

Back to Table of Contents

Here you will use a different dataset consisting of word-frequency distributions for 1,000 Twitter users. These Twitter users use language in very different ways, and were classified by hand according to the criteria:

0: Human, where only basic human-human communication is observed.

1: Cyborg, where language is primarily borrowed from other sources (e.g., jobs listings, classifieds postings, advertisements, etc...).

2: Robot, where language is formulaically derived from unrelated sources (e.g., weather/seismology, police/fire event logs, etc...).

3: Spammer, where language is replicated to high multiplicity (e.g., celebrity obsessions, personal promotion, etc...)

Check out the preprints of recent research, which spawned this dataset:

- http://arxiv.org/abs/1505.04342 (http://arxiv.org/abs/1505.04342)
- http://arxiv.org/abs/1508.01843 (http://arxiv.org/abs/1508.01843)

The main data lie in the accompanying file:

- topUsers_Apr-Jul_2014_1000-words.txt (https://www.dropbox.com/s/6129k2urvbvobkr/topUsers_Apr-Jul_2014_1000-words.txt?dl=0)

and are of the form:

USERID,CODE,TOTAL,WORD1_COUNT,WORD2_COUNT,... . .

where

USERID = unique user identifier CODE = 0/1/2/3 class code TOTAL = sum of the word counts

Using this data, you will implement a 1000-dimensional K-means algorithm in MrJob on the users by their 1000-dimensional word stripes/vectors using several centroid initializations and values of K.

Note that each "point" is a user as represented by 1000 words, and that word-frequency distributions are generally heavy-tailed power-laws (often called Zipf distributions), and are very rare in the larger class of discrete, random distributions. For each user you will have to normalize by its "TOTAL" column. **Try several parameterizations and initializations** :

- (A) K=4 uniform random centroid-distributions over the 1000 words (generate 1000 random numbers and normalize the vectors)
- (B) K=2 perturbation-centroids, randomly perturbed from the aggregated (user-wide) distribution
- (C) K=4 perturbation-centroids, randomly perturbed from the aggregated (user-wide) distribution
- (D) K=4 "trained" centroids, determined by the sums across the classes. Use use the (row-normalized) class-level aggregates as 'trained' starting centroids (i.e., the training is already

done for you!). Note that you do not have to compute the aggregated distribution or the class-aggregated distributions, which are rows in the auxiliary file:

- topUsers_Apr-Jul_2014_1000-words_summaries.txt (https://www.dropbox.com/s/w4oklbsoqefou3b/topUsers_Apr-Jul_2014_1000-words_summaries.txt?dl=0)

Row 1: Words Row 2: Aggregated distribution across all classes Row 3-6 class-aggregated distributions for clases 0-3 For (A), we select 4 users randomly from a uniform distribution [1,...,1,000] For (B), (C), and (D) you will have to use data from the auxiliary file:

- topUsers_Apr-Jul_2014_1000-words_summaries.txt (https://www.dropbox.com/s/w4oklbsoqefou3b/topUsers_Apr-Jul_2014_1000-words_summaries.txt?dl=0)

This file contains 5 special word-frequency distributions:

- (1) The 1000-user-wide aggregate, which you will perturb for initializations in parts (B) and (C), and
- (2-5) The 4 class-level aggregates for each of the user-type classes (0/1/2/3)

In parts (B) and (C), you will have to perturb the 1000-user aggregate (after initially normalizing by its sum, which is also provided). So if in (B) you want to create 2 perturbations of the aggregate, start with (1), normalize, and generate 1000 random numbers uniformly from the unit interval (0,1) twice (for two centroids), using:

```
In [13]: from numpy import random
         numbers = random.sample(1000)
```

Take these 1000 numbers and add them (component-wise) to the 1000-user aggregate, and then renormalize to obtain one of your aggregate-perturbed initial centroids.

In [349]:
```
################################################################################
##Geneate random initial centroids around the global aggregate
##Part (B) and (C) of this question
################################################################################
def startCentroidsBC(k):
    import re
    counter = 0
    for line in open("topUsers_Apr-Jul_2014_1000-words_summaries.txt").readl
        if counter == 1:
            data = re.split(",",line)
            globalAggregate = [float(data[i+3])/float(data[2]) for i in rang
        counter += 1
    #perturb the global aggregate for the four initializations
    centroids = []
    for i in range(k):
        rndpoints = random.sample(1000)
        peturpoints = [rndpoints[n]/10+globalAggregate[n] for n in range(10
        centroids.append(peturpoints)
        total = 0
        for j in range(len(centroids[i])):
            total += centroids[i][j]
        for j in range(len(centroids[i])):
            centroids[i][j] = centroids[i][j]/total
    return centroids
```

For experiments A, B, C and D and iterate until a threshold (try 0.001) is reached. After convergence, print out a summary of the classes present in each cluster. In particular, report the composition as measured by the total portion of each class type (0-3) contained in each cluster, and discuss your findings and any differences in outcomes across parts A-D.

K-Means

K-means is a clustering method that aims to find the positions μi,i=1...k of the clusters that minimize the distance from the data points to the cluster. K-means clustering solves:

$$\arg\min_c \sum_{i=1}^{k} \sum_{x \in c_i} d(x, \mu_i) = \arg\min_c \sum_{i=1}^{k} \sum_{x \in c_i} \|x - \mu_i\|_2^2$$

where c_i is the set of points that belong to cluster i. The K-means clustering uses the square of the Euclidean distance $d(x, \mu_i) = \|x - \mu_i\|_2^2$. This problem is not trivial (in fact it is NP-hard), so the K-means algorithm only hopes to find the global minimum, possibly getting stuck in a different solution.

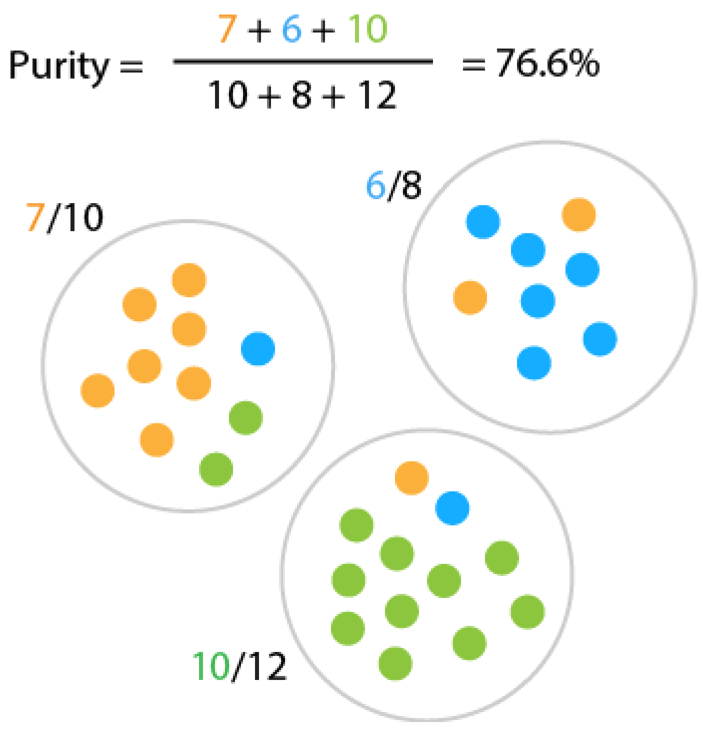
K-means algorithm

The Lloyd's algorithm, mostly known as k-means algorithm, is used to solve the k-means clustering problem and works as follows. First, decide the number of clusters k. Then:

1. Initialize the center of the clusters	$\mu_i =$ some value , $i = 1, \ldots, k$

2. Attribute the closest cluster to each data point	$c_i = \{j : d(x_j, \mu_i) \leq d(x_j, \mu_l), l \neq i, j = 1, \ldots, n\}$
3. Set the position of each cluster to the mean of all data points belonging to that cluster	$\mu_i = \frac{1}{\lvert c_i \rvert} \sum_{j \in c_i} x_j, \forall i$
4. Repeat steps 2-3 until convergence	
Notation	$\lvert c \rvert$ = number of elements in c

Calculating purity



In [621]:
```
%%writefile Kmeans.py
#!/usr/bin/env python
#START STUDENT CODE45
from numpy import argmin, array, random
import numpy as np
from sklearn import preprocessing
from mrjob.job import MRJob
from mrjob.step import MRStep
from itertools import chain
import os

# find the nearest centroid for data point
def MinDist(datapoint, centroid_points):
    datapoint = array(datapoint)
    centroid_points = array(centroid_points)
    diff = datapoint - centroid_points
    diffsq = diff*diff
    # Get the nearest centroid for each instance
    minidx = argmin(list(diffsq.sum(axis = 1)))
    return minidx

# check whether centroids converge
def stop_criterion(centroid_points_old, centroid_points_new,T):
    oldvalue = list(chain(*centroid_points_old))
    newvalue = list(chain(*centroid_points_new))
    Diff = [abs(x-y) for x, y in zip(oldvalue, newvalue)]
    Flag = True
    for i in Diff:
        if(i>T):
            Flag = False
            break
    return Flag

class MRKmeans(MRJob):
    centroid_points=[]
    def steps(self):
        return [
            MRStep(
                mapper_init = self.mapper_init,
                mapper=self.mapper,
                combiner = self.combiner,
                reducer=self.reducer
            )
        ]


    #load centroids info from file
    def mapper_init(self):
        print "Current path:", os.path.dirname(os.path.realpath(__file__))
        self.centroid_points = [map(float,s.split('\n')[0].split(',')) for s
        print "Centroids: ", self.centroid_points

    def mapper(self, _, line):
        words = (map(float,line.split(',')))
        norm_words = np.array([float(x) / int(words[2])  for x in words[3:]]
        yield int(MinDist(norm_words,self.centroid_points)), (list(norm_word
    def combiner(self, idx, inputdata):
        d_code = {}
        sumx = []

        #combine counts per code
        for x,code in inputdata:
            for key, value in code.iteritems():
                d_code[key] = d_code.get(key, 0) + value

            # convert list to an array
            x = np.array(x)

            # aggregate normalized counts
            if len(sumx) == 0:
                sumx = np.zeros(x.size)
            sumx += x

        yield idx,(list(sumx),d_code)

    # aggregate sum for each cluster and then calculate the new centroids
    def reducer(self, idx, inputdata):
        d_code = {}
        sumx = []

        #combine counts per code
        for x,code in inputdata:
            for key, value in code.iteritems():
                d_code[key] = d_code.get(key, 0) + value

            # convert list to an array
            x = np.array(x)

            # aggregate normalized counts
            if len(sumx) == 0:
                sumx = np.zeros(x.size)
            sumx += x

        # new centroids
        centroids = sumx / sum(d_code.values())

        yield idx,(list(centroids),d_code)

if __name__ == '__main__':
    MRKmeans.run()
```

```
Overwriting Kmeans.py
```

4.5A Answer

In [622]:
```
%%writefile kmeans_runner.py
#%reload_ext autoreload
#%autoreload 2
#!/usr/bin/env python
#START STUDENT CODE45_RUNNER
import numpy as np
import sys
from Kmeans import MRKmeans, stop_criterion

# set the randomizer seed so results are the same each time.
np.random.seed(0)

# define mrjob runner
mr_job = MRKmeans(args=["topUsers_Apr-Jul_2014_1000-words.txt", '--file=Cent


centroid_points = []
k = 4
class_codes = {'0':'Human', '1':'Cyborg', '2':'Robot', '3':'Spammer'}

# write initial centroids to file
centroid_points = np.random.uniform(size=[k, 1000])
with open('Centroids.txt', 'w+') as f:
    f.writelines(','.join(str(j) for j in i) + '\n' for i in centroid_points
f.close()

# Update centroids iteratively
i = 1
while(1):
    # save previous centoids to check convergency
    centroid_points_old = centroid_points
    with mr_job.make_runner() as runner:
        runner.run()
        centroid_points = []
        clusters = {}
        # stream_output: get access of the output
        for line in runner.stream_output():
            key,value =  mr_job.parse_output_line(line)
            centroid, codes = value
            centroid_points.append(centroid)
            clusters[key] = codes

    # Update the centroids for the next iteration
    with open('Centroids.txt', 'w') as f:
        f.writelines(','.join(str(j) for j in i) + '\n' for i in centroid_po

    print "\n"
    i = i + 1
    max_class={}
    if(stop_criterion(centroid_points_old,centroid_points,0.01)):
        print "Centroids\n"
        print centroid_points
        print "\n\n\n"
        print "Breakdown by class code:"
        for cluster_id, cluster in clusters.iteritems():
            max_class[cluster_id] = max(cluster.values())
            print "Cluster ID:", cluster_id
            print "Human:", cluster.get('0',0)
            print "Cyborg:", cluster.get('1',0)
            print "Robot:", cluster.get('2',0)
            print "Spammer:", cluster.get('3',0)
            print "\n"
        print "purity = ", sum(max_class.values())/1000.0*100
        break
```

```
Overwriting kmeans_runner.py
```

In [623]:
```
!python kmeans_runner.py
```

4.5B Answer

In [626]:
```
%%writefile kmeans_runner45B.py
#!/usr/bin/env python
#START STUDENT CODE45_RUNNER
import numpy as np
import sys
from Kmeans import MRKmeans, stop_criterion

# set the randomizer seed so results are the same each time.
np.random.seed(0)

# define mrjob runner
mr_job = MRKmeans(args=["topUsers_Apr-Jul_2014_1000-words.txt", '--file=Cent

#define global variables
centroid_points = []
k = 2
class_codes = {'0.0':'Human', '1.0':'Cyborg', '2.0':'Robot', '3.0':'Spammer'

#function to initialize centroids
def startCentroidsBC(k):
    import re
    counter = 0
    for line in open("topUsers_Apr-Jul_2014_1000-words_summaries.txt").readl
        if counter == 1:
            data = re.split(",",line)
            globalAggregate = [float(data[i+3])/float(data[2]) for i in rang
        counter += 1
    #perturb the global aggregate for the four initializations
    centroids = []
    for i in range(k):
        rndpoints = np.random.sample(1000)
        peturpoints = [rndpoints[n]/10+globalAggregate[n] for n in range(10
        centroids.append(peturpoints)
        total = 0
        for j in range(len(centroids[i])):
            total += centroids[i][j]
        for j in range(len(centroids[i])):
            centroids[i][j] = centroids[i][j]/total
    return centroids

# write initial centroids to file
centroid_points = startCentroidsBC(k)
with open('Centroids.txt', 'w+') as f:
    f.writelines(','.join(str(j) for j in i) + '\n' for i in centroid_points
f.close()

# Update centroids iteratively
i = 0
while(1):
    # save previous centoids to check convergency
    centroid_points_old = centroid_points[:]
    print "iteration"+str(i)+":"
    with mr_job.make_runner() as runner:
        runner.run()
        centroid_points = []
        clusters = {}
        # stream_output: get access of the output
        for line in runner.stream_output():
            key,value =  mr_job.parse_output_line(line)
            centroid, codes = value
            centroid_points.append(centroid)
            clusters[key] = codes

    # Update the centroids for the next iteration
    with open('Centroids.txt', 'w') as f:
        f.writelines(','.join(str(j) for j in i) + '\n' for i in centroid_po

    #print output of clusters and centroids
    print "\n"
    i = i + 1
    max_class={}
    if(stop_criterion(centroid_points_old,centroid_points,0.01)):
        print "Centroids\n"
        print centroid_points
        print "\n\n\n"
        print "Breakdown by class code:"
        for cluster_id, cluster in clusters.iteritems():
            max_class[cluster_id] = max(cluster.values())
            print "Cluster ID:", cluster_id
            print "Human:", cluster.get('0',0)
            print "Cyborg:", cluster.get('1',0)
            print "Robot:", cluster.get('2',0)
            print "Spammer:", cluster.get('3',0)
            print "\n"
        print "purity = ", sum(max_class.values())/1000.0*100
        break
```

```
Overwriting kmeans_runner45B.py
```

In [627]:
```
!python ./kmeans_runner45B.py
```

4.5C Answer

In [628]:
```
%%writefile kmeans_runner45C.py
#!/usr/bin/env python
#START STUDENT CODE45_RUNNER
import numpy as np
import sys
from Kmeans import MRKmeans, stop_criterion

# set the randomizer seed so results are the same each time.
np.random.seed(0)

# define mrjob runner
mr_job = MRKmeans(args=["topUsers_Apr-Jul_2014_1000-words.txt", '--file=Cent

#define global variables
centroid_points = []
k = 4
class_codes = {'0.0':'Human', '1.0':'Cyborg', '2.0':'Robot', '3.0':'Spammer'

#function to initialize centroids
def startCentroidsBC(k):
    import re
    counter = 0
    for line in open("topUsers_Apr-Jul_2014_1000-words_summaries.txt").readl
        if counter == 1:
            data = re.split(",",line)
            globalAggregate = [float(data[i+3])/float(data[2]) for i in rang
        counter += 1
    #perturb the global aggregate for the four initializations
    centroids = []
    for i in range(k):
        rndpoints = np.random.sample(1000)
        peturpoints = [rndpoints[n]/10+globalAggregate[n] for n in range(10
        centroids.append(peturpoints)
        total = 0
        for j in range(len(centroids[i])):
            total += centroids[i][j]
        for j in range(len(centroids[i])):
            centroids[i][j] = centroids[i][j]/total
    return centroids

# write initial centroids to file
centroid_points = startCentroidsBC(k)
with open('Centroids.txt', 'w+') as f:
    f.writelines(','.join(str(j) for j in i) + '\n' for i in centroid_points
f.close()

# Update centroids iteratively
i = 0
while(1):
    # save previous centoids to check convergency
    centroid_points_old = centroid_points[:]
    print "iteration"+str(i)+":"
    with mr_job.make_runner() as runner:
        runner.run()
        centroid_points = []
        clusters = {}
        # stream_output: get access of the output
        for line in runner.stream_output():
            key,value =  mr_job.parse_output_line(line)
            centroid, codes = value
            centroid_points.append(centroid)
            clusters[key] = codes

    # Update the centroids for the next iteration
    with open('Centroids.txt', 'w') as f:
        f.writelines(','.join(str(j) for j in i) + '\n' for i in centroid_po

    #print output of clusters and centroids
    print "\n"
    i = i + 1
    max_class={}
    if(stop_criterion(centroid_points_old,centroid_points,0.01)):
        print "Centroids\n"
        print centroid_points
        print "\n\n\n"
        print "Breakdown by class code:"
        for cluster_id, cluster in clusters.iteritems():
            max_class[cluster_id] = max(cluster.values())
            print "Cluster ID:", cluster_id
            print "Human:", cluster.get('0',0)
            print "Cyborg:", cluster.get('1',0)
            print "Robot:", cluster.get('2',0)
            print "Spammer:", cluster.get('3',0)
            print "\n"
        print "purity = ", sum(max_class.values())/1000.0*100
        break
```

```
Writing kmeans_runner45C.py
```

In [629]:

```
!python ./kmeans_runner45C.py
```

HW4.6 (OPTIONAL) Scaleable K-MEANS++

Back to Table of Contents

Over half a century old and showing no signs of aging, k-means remains one of the most popular data processing algorithms. As is well-known, a proper initialization of k-means is crucial for obtaining a good final solution. The recently proposed k-means++ initialization algorithm achieves this, obtaining an initial set of centers that is provably close to the optimum solution. A major downside of the k-means++ is its inherent sequential nature, which limits its applicability to massive data: one must make k passes over the data to find a good initial set of centers. The paper listed below shows how to drastically reduce the number of passes needed to obtain, in parallel, a good initialization. This is unlike prevailing efforts on parallelizing k-means that have mostly focused on the post-initialization phases of k-means. The proposed initialization algorithm k-means|| obtains a nearly optimal solution after a logarithmic number of passes; the paper also shows that in practice a constant number of passes suffices. Experimental evaluation on realworld large-scale data demonstrates that k-means|| outperforms k-means++ in both sequential and parallel settings.

Read the following paper entitled "Scaleable K-MEANS++" located at:

http://theory.stanford.edu/~sergei/papers/vldb12-kmpar.pdf (http://theory.stanford.edu/~sergei/papers/vldb12-kmpar.pdf)

In MrJob, implement K-MEANS|| and compare with a random initializtion when used in conjunction with the kmeans algorithm as an initialization step for the 2D dataset generated using code in the following notebook:

https://www.dropbox.com/s/lbzwmyv0d8rocfq/MrJobKmeans.ipynb?dl=0 (https://www.dropbox.com/s/lbzwmyv0d8rocfq/MrJobKmeans.ipynb?dl=0)

Plot the initialation centroids and the centroid trajectory as the K-MEANS|| algorithms iterates. Repeat this for a random initalization (i.e., pick a training vector at random for each inital centroid) of the kmeans algorithm. Comment on the trajectories of both algorithms. Report on the number passes over the training data, and time required to run both clustering algorithms. Also report the rand index score for both algorithms and comment on your findings.

4.6.1 (OPTIONAL) Apply K-MEANS||

Back to Table of Contents

Apply your implementation of K-MEANS|| to the dataset in HW 4.5 and compare to the a random initalization (i.e., pick a training vector at random for each inital centroid)of the kmeans algorithm. Report on the number passes over the training data, and time required to run all clustering algorithms. Also report the rand index score for both algorithms and comment on your findings.

```
In [ ]:
```

HW4.7 (OPTIONAL) Canopy Clustering

Back to Table of Contents

An alternative way to intialize the k-means algorithm is the canopy clustering. The canopy clustering algorithm is an unsupervised pre-clustering algorithm introduced by Andrew McCallum, Kamal Nigam and Lyle Ungar in 2000. It is often used as preprocessing step for the K-means algorithm or the Hierarchical clustering algorithm. It is intended to speed up clustering operations on large data sets, where using another algorithm directly may be impractical due to the size of the data set.

For more details on the Canopy Clustering algorithm see:

https://en.wikipedia.org/wiki/Canopy_clustering_algorithm (https://en.wikipedia.org/wiki/Canopy_clustering_algorithm)

Plot the initialation centroids and the centroid trajectory as the Canopy Clustering based K-MEANS algorithm iterates. Repeat this for a random initalization (i.e., pick a training vector at random for each inital centroid) of the kmeans algorithm. Comment on the trajectories of both algorithms. Report on the number passes over the training data, and time required to run both clustering algorithms. Also report the rand index score for both algorithms and comment on your findings.

```
In [ ]:
```

4.7.1 (OPTIONAL) Apply Canopy Clustering based K-MEANS

Back to Table of Contents

Apply your implementation Canopy Clustering based K-MEANS algorithm to the dataset in HW 4.5 and compare to the a random initalization (i.e., pick a training vector at random for each inital centroid)of the kmeans algorithm. Report on the number passes over the training data, and time required to run both clustering algorithms. Also report the rand index score for both algorithms and comment on your findings.

```
In [ ]:
```

Back to Table of Contents

------- END OF HOWEWORK --------