Homework #4

Student: Alex Smith
Course: W261 - Machine Learning at Scale
Professor: Jimi Shanahan
Due Date: June 10

Libraries

The following libraries must be installed before running the below code. They can all be installed through Pip (https://github.com/pypa/pip).

- Scikit Learn (http://scikit-learn.org/stable/)
- Numpy (http://www.numpy.org/)
- Regular Expression (https://docs.python.org/2/library/re.html)
- Pretty Table (https://pypi.python.org/pypi/PrettyTable)
- Random (https://docs.python.org/2/library/random.html)
- Datetime (https://docs.python.org/2/library/datetime.html)

HW 4.0.

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 MapReduce framework. It is a python package for running streaming Hadoop jobs. It was developed by Yelp to assist with producing multi-step jobs. MrJob provides a pythonic way to deal with Hadoop streaming. It's main advantage over Hadoop MapReduce is that it can schedule multiple jobs in succession. It's major disadvantage over Hadoop MapReduce is that it does not serialization of inputs/outputs in binary.
We now go over a variety of the mrjob functions:

- mapper_init: defines an action to be run before the mapper processes any data
- mapper_final: defines an action to be run after the mapper process the input
- combiner_final: defines an action for the combiner after it reaches the end of its input
- reducer_final: defines an action to be run when the reducer finishes processing its data

HW 4.1

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?

We can think of serialization as the format of the input and output data. Formally, "serialization is the process of turning structured objects into a byte stream for transmission over a network or for writing to persistent storage" (Async 4.9).By default, MrJob supports a number of protocols: RawProtocol, JSONProtocol, PickleProtocol, and ReprProtocol. It accepts as input raw text and JSON files. MrJob does not support a binary serialization scheme. Binary serialization schemes can be helpful in reducing the amount of data transferred between nodes. This can make text processing slow as data is serialized and deserialized.

HW 4.2:

Recall the Microsoft logfiles data from the async lecture. The logfiles described are located at:
[https://kdd.ics.uci.edu/databases/msweb/msweb.html (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/)](http://archive.ics.uci.edu/ml/machine-learning-databases/anonymous/)

This dataset records which areas (Vroots) of www.microsoft.com each user visited in a one-week timeframe in Feburary 1998.
Here, you must preprocess the data on a single node (i.e., not on a cluster of nodes) from the 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

to the format:
V,1000,1,C, 10001
V,1001,1,C, 10001
V,1002,1,C, 10001
Write the python code to accomplish this.

Function to consolidate a Microsoft log file

In [1]:
```
def Consolidate(filepath):
    """Function takes as input a file path.
    Returns a modified file in the same directory.
    Consolidates the information so that each
    record includes both the visitor id and
    the page id"""

    # open the file
    with open(filepath,"r") as myfile:

        # create a new file name for where
        # we will return our output
        filepath_new = filepath + "_mod"

        # open this new file
        with open(filepath_new,"w") as mynewfile:

            # set the current visitor
            visitor = None

            # loop through each line
            for line in myfile.readlines():

                # split the line by the commas
                line = line.split(",")
                category = line[0].strip()

                # if the category is a vistor id
                # or a visit id, then grab the
                # rest of the info
                if category == "C" or \
                category == "V":
                    record_id = int(line[1].replace("\"",""))
                    simple = int(line[2].strip())

                # if this is the line that
                # identifies the visitor
                if category == "C":

                    # set the visitor
                    visitor = record_id

                # else we are dealing with a
                # page visit
                elif category == "V":

                    # write to the new file with
                    # visit id and the visitor id
                    info = "V," + str(record_id) \
                    + "," + str(simple) + ",C," \
                    + str(visitor)+"\n"
                    mynewfile.write(info)
```

In [2]:
```
# put our log file through this function
Consolidate("anonymous-msweb.data")
```

In [3]:
```
# sample the top of the output file to gut
# check if our program worked
!head anonymous-msweb.data_mod
```

```
V,1000,1,C,10001
V,1001,1,C,10001
V,1002,1,C,10001
V,1001,1,C,10002
V,1003,1,C,10002
V,1001,1,C,10003
V,1003,1,C,10003
V,1004,1,C,10003
V,1005,1,C,10004
V,1006,1,C,10005
```

HW 4.3

Find the 5 most frequently visited pages using MrJob from the output of 4.2 (i.e., transfromed log file).

Write the MRJob class

In [1]:
```
%%writefile mr_pagevisit.py
# import MrJob
from mrjob.job import MRJob

# create the class
class MRPageVisit(MRJob):
    """A page visit class implemented
    in MRJob"""

    def __init__(self, *args, **kwargs):
        # gather the arguments (i.e. the files
        # we want to perform the function on)
        super(MRPageVisit, self).__init__(*args, **kwargs)

    def mapper(self, _, line):
        """takes the words from the input where
        the value is the text of the line"""

        # split the line based on commas
        line = line.split(",")

        # grab the page visited
        page = int(line[1])

        # yield the page with a simple count
        # of 1
        yield page, 1

    def reducer(self, key, values):
        """outputs the sum of visits for each
        page visited"""

        # output the sum of page views
        yield key, sum(values)
```

```
Overwriting mr_pagevisit.py
```

Use a runner to run the MRJob within the notebook

In [7]:
```
# import the MRJob that we created
from mr_pagevisit import MRPageVisit

# set the data that we're going to pull
mr_job = MRPageVisit(args=['anonymous-msweb.data_mod'])

# create the runner and run it
with mr_job.make_runner() as runner:
    runner.run()

    # create a file to write to
    with open("HW4.3_Output","w") as myfile:

        # stream_output: get access of the output
        for line in runner.stream_output():

            # write the output to a file
            info=str(mr_job.parse_output_line(line))+"\n"
            myfile.write(info)
```

Show the top 5 most frequently visited pages

In [14]:
```
!cat HW4.3_Output | sort -k2nr > temp
!head -5 temp
!rm temp
```

```
(1008, 10836)
(1034, 9383)
(1004, 8463)
(1018, 5330)
(1017, 5108)
```

HW 4.4

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.

Function to gather the webpage urls based on the webpage IDs

In [17]:
```
def GatherWeb(filename):
    """Takes as input the file path to a
    Microsoft log file. Gather the URLs and
    webpage id combinations. Returns a file
    that matches each webpage to its id"""

    # open the file
    with open(filename, "r") as myfile:

        # the name of the new file
        newfile = "MS_webpages"

        # open the new file to write
        with open(newfile,"w") as mynewfile:

            # loop through each line in the file
            for line in myfile.readlines():

                # split the line by commas
                line = line.split(",")

                # set the category
                category = line[0]

                # if the category is the description
                # of the webpage
                if category == "A":

                    # set the web_id and the
                    # web_url
                    web_id = line[1]
                    web_url = line[3].replace("\"","")

                    # write to the new file
                    info = str(web_id) + "," \
                    + str(web_url) + "\n"
                    mynewfile.write(info)
```

In [18]:
```
GatherWeb("anonymous-msweb.data")
!head MS_webpages
```

```
1287,International AutoRoute
1288,library
1289,Master Chef Product Information
1297,Central America
1215,For Developers Only Info
1279,Multimedia Golf
1239,Microsoft Consulting
1282,home
1251,Reference Support
1121,Microsoft Magazine
```

MRJob class to calculate the most frequent visitor for each webpage

```
%%writefile mr_freqvisit.py
# import MRJob
from mrjob.job import MRJob

# create the class
class MRFreqVisit(MRJob):
    """MRJob class that identifies the most
    frequent visitor for each webpage"""

    def __init__(self, *args, **kwargs):

        # allow us to take a file as input
        super(MRFreqVisit,self).__init__(*args, **kwargs)

        # create a dictionary to hold the information
        # that matches each web id to it's url
        self.urls = {}

        # gather the webpage name data
        with open('MS_webpages','r') as myfile:

            # read through each line
            for line in myfile.readlines():

                # gather the id and url
                line = line.split(",")
                web_id = line[0].strip()
                web_url = line[1].strip()

                # add the id and url to
                # the dictionary
                self.urls[web_id] = web_url


    def mapper(self, _, line):

        # break the line up
        line = line.split(",")

        # gather the website and the visitor
        site = line[1]
        visitor = line[4]

        # yield the site with the visitor
        # and a count of 1
        yield site,(visitor,1)

    def reducer(self, key, values):

        # create a dictionary for this site
        visitor_counts = {}

        # convert the values to a tuple
        visitors = tuple(values)

        # loop through the values for each site
        for item in visitors:
```

```
            # split into the visitor id and
            # the count
            visitor_id = item[0]
            visitor_count = item[1]

            # check to see if this visitor is
            # already in the dictionary, if
            # it's not, add it
            if visitor_id not in \
            visitor_counts.keys():
                visitor_counts[visitor_id] = 0

            # add the count to the dictionary
            visitor_counts[visitor_id] = \
            visitor_counts[visitor_id] + \
            visitor_count

        # set a max place holder
        max_visitor = None
        max_count = 0

        # loop through the keys and update the
        # max visitor
        for visitor in visitor_counts:

            # check to see if it's a new max
            if visitor_counts[visitor] > max_count:
                max_count = visitor_counts[visitor]
                max_visitor = visitor


        # let's format it nicely by grabbing
        # everything we need
        url = self.urls[key]
        info = url + "    " + max_visitor + "    " \
        + str(max_count)

        # yield the page, the visitor, and
        # the count
        yield key,info
```

```
Overwriting mr_freqvisit.py
```

Create a runner to run the MRJob within the notebook

In [2]:
```
# import the MRJob that we created
from mr_freqvisit import MRFreqVisit

# set the data that we're going to pull
mr_job = MRFreqVisit(args=['anonymous-msweb.data_mod','--file=MS_webpage
s'])

# create the runner and run it
with mr_job.make_runner() as runner:
    runner.run()

    # create a file to write to
    with open("HW4.4_Output","w") as myfile:

        # stream_output: get access of the output
        for line in runner.stream_output():

            # write the output to a file
            info=str(mr_job.parse_output_line(line))+"\n"
            myfile.write(info)
```

Show the most frequent visitors for a couple of webpages

In [6]:
```
!echo Webpage.ID___URL___Visitor.ID___Visits
!head HW4.4_Output
```

```
Webpage.ID___URL___Visitor.ID___Visits
('1000', 'regwiz    36585    1')
('1001', 'Support Desktop    23995    1')
('1002', 'End User Produced View    35235    1')
('1003', 'Knowledge Base    22469    1')
('1004', 'Microsoft.com Search    35540    1')
('1005', 'Norway    10004    1')
('1006', 'misc    27495    1')
('1007', 'International IE content    19492    1')
('1008', 'Free Downloads    35236    1')
('1009', 'Windows Family of OSs    22504    1')
```

HW 4.5 Clustering Tweet Dataset

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

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 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

- *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
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:

```
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.

```
###########################################
## Geneate random initial centroids around the global aggregate
## Part (B) and (C) of this question
###########################################

def startCentroidsBC(k):
    counter = 0
    for line in open("topUsers_Apr-Jul_2014_1000-words_summaries.txt").readl
ines():
        if counter == 2:
            data = re.split(",",line)
            globalAggregate = [float(data[i+3])/float(data[2]) for i in rang
e(1000)]
        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(100
0)]
        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.

Part A. K=4 uniform random centroid-distributions over the 1000 words

Normalize the data

Every part of the question requires us to normalize the data. Rather than doing this for each step, let's do it once at the beginning and write to an output file.

In [1]:
```
# import pandas to allow us to act efficiently
# with this data set
import pandas as pd

# read in the twitter data
raw_data = \
pd.read_csv(\
            "topUsers_Apr-Jul_2014_1000-words.txt",\
            header=None)

# divide each word count by the total and rename
# the file to reflect it's normalized state
raw_data.ix[:,3:] = raw_data.ix[:,3:].\
div(raw_data[2],'index')
norm_data = raw_data

# write the file to the local drive and
# show the first couple lines
norm_data.to_csv('twitter_users_norm.txt',header=False,index=False)
print norm_data.head()
```

```
            0  1        2         3         4         5         6   \
0  1180025371  2  1724608  0.043808  0.000480  0.033401  0.004133
1   284534859  2   827765  0.120714  0.000000  0.017442  0.034623
2  1602852614  2   987334  0.000000  0.002734  0.000000  0.000000
3  2361533634  2   416584  0.134612  0.000132  0.000007  0.000000
4   485013829  1   530484  0.102629  0.000019  0.000000  0.000000

         7         8         9     ...          993  994  995  996  997
 \
0  0.002483  0.026484  0.038740  ...    0.000102  0.0  0.0  0.0  0.0

1  0.009022  0.031469  0.033303  ...    0.000000  0.0  0.0  0.0  0.0

2  0.000000  0.000000  0.000000  ...    0.000000  0.0  0.0  0.0  0.0

3  0.000000  0.000000  0.000007  ...    0.000000  0.0  0.0  0.0  0.0

4  0.000000  0.000000  0.015812  ...    0.000000  0.0  0.0  0.0  0.0


        998  999  1000  1001  1002
0  0.000070  0.0   0.0   0.0   0.0
1  0.000000  0.0   0.0   0.0   0.0
2  0.000000  0.0   0.0   0.0   0.0
3  0.001359  0.0   0.0   0.0   0.0
4  0.000000  0.0   0.0   0.0   0.0

[5 rows x 1003 columns]
```

Create the initial centroids

In [2]:
```
# import libraries to help us get started
import numpy as np
import csv
import pandas as pd

# begin by creating the centroids
# define how many centroids we need
# and create a list to store them
K = 4
centroid_index = []

# grab the number of users
with open('twitter_users_norm.txt','r') as myfile:
    lines = myfile.readlines()
    num_users = len(lines)

# loop through each possible centroid
for point in range(K):

    # get the number of a random user
    _user = \
    np.random.randint(0,num_users-1)

    # add that user to our list
    centroid_index.append(_user)


# create an array to hold the centroid values
centroids = []

# pull the centroid values from the randomly
# selected users and write it to a local file
with open('centroids.txt','w') as myfile:

    # loop through our indexes
    for index in centroid_index:

        # set the centroid as the line from
        # the 3rd element on
        centroid = lines[index].split(",")[3:]
        centroids.append(centroid)

# convert our array to a pandas data frame
# and write that data frame to an output file
centroids = pd.DataFrame(centroids)
centroids.to_csv('centroids.txt',header=False,index=False)

# print the first couple lines of our data frame
print centroids.head()
```

```
                 0                  1                   2                   
 3    \
0  0.00573363807452  0.00695318771852    0.0125332853702    0.025685119
3009
1   0.0920348952663     0.01482843389  2.73903188918e-05  6.16282175065
e-05
2    0.039549886789   0.0524525253445   0.00240075549841   0.0032313486
3293
3  0.00573363807452  0.00695318771852    0.0125332853702    0.025685119
3009

                4                  5                 6                  
 7    \
0  0.0480756998153     0.016718358609  0.00196970428115   0.00099581931
363
1              0.0  2.73903188918e-05   0.0155405821812    0.0307319377
966
2  0.0204348667069  0.000785081181945  0.00225284164525  0.000614411359
783
3  0.0480756998153     0.016718358609  0.00196970428115   0.00099581931
363

                8                 9    ...                   990  991  \
0              0.0  0.0455400894044  ...                   0.0  0.0   
1  0.0920348952663               0.0  ...     2.05427391688e-05  0.0   
2              0.0  0.0313577353252  ...     3.41339644324e-05  0.0   
3              0.0  0.0455400894044  ...                   0.0  0.0   

                  992                993                 994          
   995  \
0  4.38686922304e-06  6.14161691226e-05  0.000127219207468  1.754747689
22e-05
1                0.0                0.0                0.0            
   0.0
2  1.13779881441e-05                0.0  1.13779881441e-05            
   0.0
3  4.38686922304e-06  6.14161691226e-05  0.000127219207468  1.754747689
22e-05

                  996  997                998    999
0  1.75474768922e-05  0.0                0.0  0.0\n
1                0.0  0.0                0.0  0.0\n
2                0.0  0.0  1.13779881441e-05  0.0\n
3  1.75474768922e-05  0.0                0.0  0.0\n

[4 rows x 1000 columns]
```

Count the members in each class and write to a file

This is useful for when we have to calculate the purities.

In [3]:
```
# import pandas to help us process the data
import pandas as pd

# open the file as a pandas dataframe
data = \
pd.read_csv(\
            "topUsers_Apr-Jul_2014_1000-words.txt",\
            header=None)

# store the counts for each class
classes = data.ix[:,1].value_counts()

# print the classes
print classes

# create a blank array to hold the classes
classes_out = []

# loop through and add to the array each element
for i in range(len(classes)):
    classes_out.append(classes[i])

# save the classes as a pandas dataframe and write
# the file to the disk
classes = classes_out
classes = pd.DataFrame(classes)
classes.to_csv('class_counts.txt',header=False,index=False)
```

```
0    752
3    103
1     91
2     54
Name: 1, dtype: int64
```

Create the MRJob class that finds the next closest centroid

```
%%writefile mr_kmeans.py
# import MRJob and some other libraries
# to help us get started
from mrjob.job import MRJob
from mrjob.step import MRStep
import numpy as np
import re
import pandas as pd

# define a function that will find which centroid
# is closest to a given point
def ClosestCentroid(point,centroid_points):
    """takes a point, a list of coordinates, and
    compares that point to each of a number of
    centroids stored in a list of lists. returns
    the index of the centroid closest to the data
    point"""

    # convert our inputs into numpy arrays
    point = np.array(point)
    centroid_points = np.array(centroid_points)

    # calculate the difference between the point
    # and each of the centroid points
    difference = point - centroid_points

    # square the difference, this will help us
    # calculate distance regardless of direction
    diff_sq = difference * difference

    # get the index of the centroid that is
    # closest to the data point
    closest_index = \
    np.argmin(list(diff_sq.sum(axis=1)))

    # return the closest index
    return int(closest_index)


# create the MRJob class
class MRKmeans(MRJob):
    """class responsible for find the nearest centroid
    to a number of data points"""

    # create an array to hold our centroid
    # points and set a value for K, number
    # of centroids
    centroid_points = []
    K=4

    # read in the class count file
    classes_pd = pd.read_csv('class_counts.txt',\
                             header=None)

    # set the class counts
    class_counts = map(float,classes_pd.values)
```

```
    # set the number of true classifications
    # and set the number of dimensions in our
    # data
    TRUTHS = 4
    DIMS = 1000

    # create an empty array to hold the counts
    # for each class
    classes = [0] * TRUTHS

    # define the steps of the job and the order in
    # which they will be executed
    def steps(self):
        return [MRStep(mapper_init=self.mapper_init,\
                       mapper=self.mapper,\
                       combiner=self.combiner,\
                       reducer=self.reducer)]


    # load the initial centroids from a
    # data file passed in
    def mapper_init(self):

        # read in the centroids data
        centroids = pd.read_csv('centroids.txt',\
                                header=None)

        # set the centroid points based on the
        # inputted file
        self.centroid_points = map(list,centroids.values)


    # takes a line of the twitter data and
    # returns the index of the closest centroid
    # and the coordinates of this point,
    # along with this point's true class
    def mapper(self, _, line):

        # get all the information for the point
        point = map(float,line.split(','))

        # get the point's true classification
        # and simplify the point to just it's
        # coordinates
        truth = int(point[1])
        point = point[3:]

        # grab the closest centroid
        closest = \
        ClosestCentroid(point,self.centroid_points)

        # create an array of zeros of the
        # length of the true classifications
        classify = [0] * self.TRUTHS

        # set the index of the truth to be 1
        classify[truth] = 1
```

```
        # yield:
        # key: the index of the closest cluster
        # value: the coordinates of the point &
        # the classification
        yield closest, (point, classify)


    # takes the output of the mapper and combines
    # the coordinate positions and updates the
    # count of points for this centroid
    def combiner(self, centroid, point_classify):

        # get the centroid value
        centroid = int(centroid)

        # set two blank arrays to hold the sums of
        # the coordinates and the sums of the true
        # classifications
        coordinates = [0] * self.DIMS
        truths = [0] * self.TRUTHS

        # convert our arrays to numpy arrays
        coordinates = np.array(coordinates)
        truths = np.array(truths)

        # loop through each point and its
        # associated classification
        for point,classify in point_classify:

            # set each element as a numpy array
            point = np.array(point)
            classify = np.array(classify)

            # sum the coordinates and
            # classification values
            coordinates = coordinates + point
            truths = truths + classify

        # convert the numpy arrays back to
        # regular arrays for the combiner's
        # output
        coordinates = list(coordinates)
        truths = list(truths)

        # yield the key as the centroid and the
        # sum of the coordinates and the sum of
        # the classifications
        yield centroid, (coordinates,truths)

    # takes the outputs of the mappers and
    # combiners and computes the aggregate
    # sums for each centroid and uses these
    # sums to calculate new centroids at
    # the centers of the clusters
    def reducer(self, centroid, point_classify):
```

```
        # get the centroid value
        centroid = int(centroid)

        # set two blank arrays to hold the sums of
        # the coordinates and the sums of the true
        # classifications
        coordinates = [0] * self.DIMS
        truths = [0] * self.TRUTHS

        # convert our arrays to numpy arrays
        coordinates = np.array(coordinates)
        truths = np.array(truths)

        # loop through each point and its
        # associated classification
        for point,classify in point_classify:

            # set each element as a numpy array
            point = np.array(point)
            classify = np.array(classify)

            # sum the coordinates and
            # classification values
            coordinates = coordinates + point
            truths = truths + classify

        # gather the complete count for the
        # centroid
        num_points = float(sum(truths))

        # calculate the new centroid and
        # convert it back to a regular list
        new_centroid = coordinates / num_points
        new_centroid = list(new_centroid)

        # print out the class breakdown
        print "Cluster #",centroid
        for index,item in enumerate(truths):
            proportion = float(item) /\
            self.class_counts[index]

            print "\tClass",index,"\t",proportion

        # yield the centroid index and the
        # coordinates of the new centroid
        yield centroid, new_centroid
```

Overwriting mr_kmeans.py

In [5]:

```
# create a test file that we used to test
# each step of the MRJob
!head -50 twitter_users_norm.txt > test.txt
```

Create a stop function to tell us when we have achieved sufficient convergence

In [6]:
```
# import the chain tool to combine lists
from itertools import chain

def stop_reached(centroids_old,\
                 centroids_new,thresh=0.5):
    """a function that compares two lists of
    centroids to determine if coordinate has
    moved a greater distance than the
    threshold, by default set to 0.5"""

    # convert the lists of centroids into a
    # single list because we don't care about
    # the context of the coordinates
    centroids_old = list(chain(*centroids_old))
    centroids_new = list(chain(*centroids_new))

    # calculate the difference between each
    # of the coordinates
    difference = [abs(old-new) for old,new in\
                  zip(centroids_old,centroids_new)]

    # set the flag for stopping to true
    # by default
    stopping = True

    # loop through each difference
    for diff in difference:

        # if the difference is greater
        # than the threshold, then break
        # out of the loop and set the
        # indicator for stopping to
        # false
        if diff > thresh:
            stopping = False
            break

    # return whether or not we reached the
    # threshold or we need to keep going
    return stopping
```

Run the MRJob in the notebook and print out the answer to part (A)

```
# import the MRJob that we created
from mr_kmeans import MRKmeans

# import pandas to help us save and load
# the centroids
import pandas as pd

# set the data that we're going to pull
mr_job = MRKmeans(args=['twitter_users_norm.txt',\
                        '--file=centroids.txt',\
                       '--file=class_counts.txt'])

# read in the centroids data to get the original
# centroids and convert it to a list
centroids = pd.read_csv('centroids.txt',\
                        header=None)
centroids = map(list,centroids.values)

# create a counter to count our iterations
# and an initial stopping indicator
iteration = 0
stopping = False

# set up a loop that runs until we tell
# it to stop
while stopping == False:

    # set the old centroids
    old_centroids = centroids[:]

    # create a new array to hold the
    # new centroid points
    new_centroids = []

    # print the iteration we are on
    print "\n*~*~*~*~*~*~*~*~*\n"
    print "Iteration:", iteration

    # create the runner and run it
    with mr_job.make_runner() as runner:
        runner.run()

        # stream_output: get access of the output
        for line in runner.stream_output():

            # set the centroid
            index,coordinates = mr_job.parse_output_line(line)

            # update the current centroid
            new_centroids.append(coordinates)

            # print out the centroid values
            print "Index:", index
            print "Coordinates sample:", coordinates[0:4]

        # set the new centroids as a regular list
        new_centroids = new_centroids[:]
```

```
        centroids = new_centroids[:]

        # convert our array to a pandas data frame
        # and write that data frame to an output file
        # and update the centroids file
        centroids_pd = pd.DataFrame(centroids)
        centroids_pd.to_csv('HW4.5_A_Output',\
                         header=False,index=False)
        centroids_pd.to_csv('centroids.txt',\
                         header=False,index=False)

        # check the stopping condition with our
        # new and old centroids
        stopping = stop_reached(old_centroids,\
                                new_centroids,thresh=0.001)

        # iterate the iteration count by 1
        iteration = iteration + 1
```

```
*~*~*~*~*~*~*~*

Iteration: 0
Cluster # 0
        Class 0         0.797872340426
        Class 1         0.032967032967
        Class 2         0.0740740740741
        Class 3         0.368932038835
Cluster # 1
        Class 0         0.0
        Class 1         0.56043956044
        Class 2         0.037037037037
        Class 3         0.0
Cluster # 2
        Class 0         0.202127659574
        Class 1         0.406593406593
        Class 2         0.888888888889
        Class 3         0.631067961165
Index: 0
Coordinates sample: [0.011329639554444023, 0.04460399573929679, 0.02673
750226515437, 0.02788322375644552]
Index: 1
Coordinates sample: [0.12472895964830379, 0.002487320646417646, 0.00083
40059164346096, 0.0008403320409112254]
Index: 2
Coordinates sample: [0.05272958802582364, 0.041476597372466326, 0.02219
005224033261, 0.02146342433378848]

*~*~*~*~*~*~*~*

Iteration: 1
Cluster # 0
        Class 0         0.957446808511
        Class 1         0.021978021978
        Class 2         0.037037037037
        Class 3         0.572815533981
Cluster # 1
        Class 0         0.0
        Class 1         0.604395604396
        Class 2         0.0555555555556
        Class 3         0.0
Cluster # 2
        Class 0         0.0425531914894
        Class 1         0.373626373626
        Class 2         0.907407407407
        Class 3         0.427184466019
Index: 0
Coordinates sample: [0.012123849884720756, 0.04754885431536065, 0.02553
344950506735, 0.027376907364399735]
Index: 1
Coordinates sample: [0.13380382094462412, 0.0025006046736551417, 0.0012
037710242803847, 0.0017795507220235507]
Index: 2
Coordinates sample: [0.079176049580193, 0.02548144435592435, 0.02470929
1630817858, 0.018690809192939747]

*~*~*~*~*~*~*~*
```

```
Iteration: 2
Cluster # 0
        Class 0         0.990691489362
        Class 1         0.032967032967
        Class 2         0.037037037037
        Class 3         0.844660194175
Cluster # 1
        Class 0         0.0
        Class 1         0.593406593407
        Class 2         0.0555555555556
        Class 3         0.0
Cluster # 2
        Class 0         0.0093085106383
        Class 1         0.373626373626
        Class 2         0.907407407407
        Class 3         0.155339805825
Index: 0
Coordinates sample: [0.013745081574133258, 0.04776693858166882, 0.02515
7575324386033, 0.02697235890854639]
Index: 1
Coordinates sample: [0.13085062239030174, 0.0025295667372873375, 0.0008
059181608256544, 0.0008011263368923851]
Index: 2
Coordinates sample: [0.1026365019243255, 0.012285139204803517, 0.027249
61480658368, 0.017826815681354538]

*~*~*~*~*~*~*~*

Iteration: 3
Cluster # 0
        Class 0         0.998670212766
        Class 1         0.032967032967
        Class 2         0.148148148148
        Class 3         0.922330097087
Cluster # 1
        Class 0         0.0
        Class 1         0.56043956044
        Class 2         0.0185185185185
        Class 3         0.0
Cluster # 2
        Class 0         0.00132978723404
        Class 1         0.406593406593
        Class 2         0.833333333333
        Class 3         0.0776699029126
Index: 0
Coordinates sample: [0.014397571611987205, 0.047388046524136675, 0.0248
6042764029234, 0.026662811484961094]
Index: 1
Coordinates sample: [0.1139868247017923, 0.002217108452177485, 0.000218
26160449335208, 0.0005053564866081719]
Index: 2
Coordinates sample: [0.12721488133856046, 0.0076977158449778645, 0.0293
9066989024009, 0.01796552846114821]

*~*~*~*~*~*~*~*
```

```
Iteration: 4
Cluster # 0
        Class 0         0.998670212766
        Class 1         0.032967032967
        Class 2         0.240740740741
        Class 3         0.95145631068
Cluster # 1
        Class 0         0.0
        Class 1         0.56043956044
        Class 2         0.0
        Class 3         0.0
Cluster # 2
        Class 0         0.00132978723404
        Class 1         0.406593406593
        Class 2         0.759259259259
        Class 3         0.0485436893204
Index: 0
Coordinates sample: [0.014566847899000618, 0.04706320932510353, 0.02476
222185169291, 0.026499363912542147]
Index: 1
Coordinates sample: [0.10931912915849412, 0.0022586353652849747, 0.0002
225412437971433, 0.0005152654373259792]
Index: 2
Coordinates sample: [0.1388927254530536, 0.007172275985086444, 0.030483
518405095462, 0.01860646128841889]

*~*~*~*~*~*~*~*

Iteration: 5
Cluster # 0
        Class 0         0.998670212766
        Class 1         0.032967032967
        Class 2         0.259259259259
        Class 3         0.95145631068
Cluster # 1
        Class 0         0.0
        Class 1         0.56043956044
        Class 2         0.0
        Class 3         0.0
Cluster # 2
        Class 0         0.00132978723404
        Class 1         0.406593406593
        Class 2         0.740740740741
        Class 3         0.0485436893204
Index: 0
Coordinates sample: [0.014600613913121866, 0.04700941754673868, 0.02477
219692305493, 0.0264735368331374]
Index: 1
Coordinates sample: [0.10931912915849412, 0.0022586353652849747, 0.0002
225412437971433, 0.0005152654373259792]
Index: 2
Coordinates sample: [0.1400383219509458, 0.00725291148778457, 0.0304483
7243827492, 0.01878083897685731]

*~*~*~*~*~*~*~*

Iteration: 6
```

```
Cluster # 0
        Class 0         0.998670212766
        Class 1         0.032967032967
        Class 2         0.259259259259
        Class 3         0.961165048544
Cluster # 1
        Class 0         0.0
        Class 1         0.56043956044
        Class 2         0.0
        Class 3         0.0
Cluster # 2
        Class 0         0.00132978723404
        Class 1         0.406593406593
        Class 2         0.740740740741
        Class 3         0.0388349514563
Index: 0
Coordinates sample: [0.014694596354347216, 0.04700762291410899, 0.02475
6628910398697, 0.026451644460645893]
Index: 1
Coordinates sample: [0.10931912915849412, 0.0022586353652849747, 0.0002
225412437971433, 0.0005152654373259792]
Index: 2
Coordinates sample: [0.14057435770089027, 0.0067870510052356, 0.0306821
973466673, 0.01891849738044094]

*~*~*~*~*~*~*~*

Iteration: 7
Cluster # 0
        Class 0         0.998670212766
        Class 1         0.032967032967
        Class 2         0.259259259259
        Class 3         0.961165048544
Cluster # 1
        Class 0         0.0
        Class 1         0.56043956044
        Class 2         0.0
        Class 3         0.0
Cluster # 2
        Class 0         0.00132978723404
        Class 1         0.406593406593
        Class 2         0.740740740741
        Class 3         0.0388349514563
Index: 0
Coordinates sample: [0.014694596354347216, 0.04700762291410899, 0.02475
6628910398697, 0.026451644460645893]
Index: 1
Coordinates sample: [0.10931912915849412, 0.0022586353652849747, 0.0002
225412437971433, 0.0005152654373259792]
Index: 2
Coordinates sample: [0.14057435770089027, 0.0067870510052356, 0.0306821
973466673, 0.01891849738044094]
```

Part B. K=2, Perturbation centroids from the aggregated data

Write the function to generate the random centroids from the aggregate data

```
# import the numpy library to help us
# with randomization
import numpy as np
import re

# generate 1000 random numbers
numbers = np.random.sample(1000)

def aggregateCentroids(k):
    """generates k centroid points from the
    aggregate data that is randomly perturbed"""

    # initalize a counter at zero
    counter = 0

    # loop through each line in the summary data
    for line in \
    open("topUsers_Apr-Jul_2014_1000-words_summaries.txt")\
    .readlines():

        # if it's the third line
        if counter == 1:

            # split the line by commas
            data = re.split(",",line)

            # calculate the global aggregate as
            # the normalized count for each word
            globalAggregate = \
            [float(data[i+3])/float(data[2]) \
             for i in range(1000)]

        # increment our line counter by 1
        counter += 1

    # create an empty array to hold the future
    # centroid points
    centroids = []

    # loop the number of centroids needed
    for i in range(k):

        # generate a set of 1000 random points
        rndpoints = np.random.sample(1000)

        # peturb the aggregate coordinates by
        # the random points generated above
        peturpoints = \
        [rndpoints[n]/10+globalAggregate[n] \
         for n in range(1000)]

        # append our preturbed centroid to the
        # list of centroids
        centroids.append(peturpoints)

        # renormalize, start by initalizing a
        # new total
```

```
        total = 0

        # total up the values of the centroid
        for j in range(len(centroids[i])):
            total += centroids[i][j]

        # renormalize the centroids by dividing
        # by the total
        for j in range(len(centroids[i])):
            centroids[i][j] = centroids[i][j]/total

    # return the new centroids
    return centroids
```

Generate the random centroids

In [3]:
```
# import pandas to help us write to csvs
import pandas as pd

# generate 2 centroids from the aggregate
# data
centroids = aggregateCentroids(2)

# write the data to a centroids file
centroids_pd = pd.DataFrame(centroids)
centroids_pd.to_csv('centroids.txt',\
                    header=False,index=False)

# read the first couple lines
print "Done. We got", len(centroids), "centroids"
```

```
Done. We got 2 centroids
```

Write the MRJob class to find the best centroids

```
%%writefile mr_kmeans.py
# import MRJob and some other libraries
# to help us get started
from mrjob.job import MRJob
from mrjob.step import MRStep
import numpy as np
import re
import pandas as pd

# define a function that will find which centroid
# is closest to a given point
def ClosestCentroid(point,centroid_points):
    """takes a point, a list of coordinates, and
    compares that point to each of a number of
    centroids stored in a list of lists. returns
    the index of the centroid closest to the data
    point"""

    # convert our inputs into numpy arrays
    point = np.array(point)
    centroid_points = np.array(centroid_points)

    # calculate the difference between the point
    # and each of the centroid points
    difference = point - centroid_points

    # square the difference, this will help us
    # calculate distance regardless of direction
    diff_sq = difference * difference

    # get the index of the centroid that is
    # closest to the data point
    closest_index = \
    np.argmin(list(diff_sq.sum(axis=1)))

    # return the closest index
    return int(closest_index)


# create the MRJob class
class MRKmeans(MRJob):
    """class responsible for find the nearest centroid
    to a number of data points"""

    # create an array to hold our centroid
    # points and set a value for K, number
    # of centroids
    centroid_points = []
    K=2

    # set the number of true classifications
    # and set the number of dimensions in our
    # data
    TRUTHS = 4
    DIMS = 1000

    # read in the class count file
```

```
    classes_pd = pd.read_csv('class_counts.txt',\
                             header=None)

    # set the class counts
    class_counts = map(float,classes_pd.values)

    # create an empty array to hold the counts
    # for each class
    classes = [0] * TRUTHS

    # define the steps of the job and the order in
    # which they will be executed
    def steps(self):
        return [MRStep(mapper_init=self.mapper_init,\
                       mapper=self.mapper,\
                       combiner=self.combiner,\
                       reducer=self.reducer)]


    # load the initial centroids from a
    # data file passed in
    def mapper_init(self):

        # read in the centroids data
        centroids = pd.read_csv('Centroids.txt',\
                                header=None)

        # set the centroid points based on the
        # inputted file
        self.centroid_points = map(list,centroids.values)


    # takes a line of the twitter data and
    # returns the index of the closest centroid
    # and the coordinates of this point,
    # along with this point's true class
    def mapper(self, _, line):

        # get all the information for the point
        point = map(float,line.split(','))

        # get the point's true classification
        # and simplify the point to just it's
        # coordinates
        truth = int(point[1])
        point = point[3:]

        # grab the closest centroid
        closest = \
        ClosestCentroid(point,self.centroid_points)

        # create an array of zeros of the
        # length of the true classifications
        classify = [0] * self.TRUTHS

        # set the index of the truth to be 1
        classify[truth] = 1
```

```
        # yield:
        # key: the index of the closest cluster
        # value: the coordinates of the point &
        # the classification
        yield closest, (point, classify)


    # takes the output of the mapper and combines
    # the coordinate positions and updates the
    # count of points for this centroid
    def combiner(self, centroid, point_classify):

        # get the centroid value
        centroid = int(centroid)

        # set two blank arrays to hold the sums of
        # the coordinates and the sums of the true
        # classifications
        coordinates = [0] * self.DIMS
        truths = [0] * self.TRUTHS

        # convert our arrays to numpy arrays
        coordinates = np.array(coordinates)
        truths = np.array(truths)

        # loop through each point and its
        # associated classification
        for point,classify in point_classify:

            # set each element as a numpy array
            point = np.array(point)
            classify = np.array(classify)

            # sum the coordinates and
            # classification values
            coordinates = coordinates + point
            truths = truths + classify

        # convert the numpy arrays back to
        # regular arrays for the combiner's
        # output
        coordinates = list(coordinates)
        truths = list(truths)

        # yield the key as the centroid and the
        # sum of the coordinates and the sum of
        # the classifications
        yield centroid, (coordinates,truths)

    # takes the outputs of the mappers and
    # combiners and computes the aggregate
    # sums for each centroid and uses these
    # sums to calculate new centroids at
    # the centers of the clusters
    def reducer(self, centroid, point_classify):
```

```
        # get the centroid value
        centroid = int(centroid)

        # set two blank arrays to hold the sums of
        # the coordinates and the sums of the true
        # classifications
        coordinates = [0] * self.DIMS
        truths = [0] * self.TRUTHS

        # convert our arrays to numpy arrays
        coordinates = np.array(coordinates)
        truths = np.array(truths)

        # loop through each point and its
        # associated classification
        for point,classify in point_classify:

            # set each element as a numpy array
            point = np.array(point)
            classify = np.array(classify)

            # sum the coordinates and
            # classification values
            coordinates = coordinates + point
            truths = truths + classify

        # gather the complete count for the
        # centroid
        num_points = float(sum(truths))

        # calculate the new centroid and
        # convert it back to a regular list
        new_centroid = coordinates / num_points
        new_centroid = list(new_centroid)

        # print out the class breakdown
        print "Cluster #",centroid
        for index,item in enumerate(truths):
            proportion = float(item) /\
            self.class_counts[index]

            print "\tClass",index,"\t",proportion

        # yield the centroid index and the
        # coordinates of the new centroid
        yield centroid, new_centroid
```

```
Overwriting mr_kmeans.py
```

Create a copy of the stop function to tell us when we have achieved sufficient convergence

We put a copy down here because we don't want to have to scroll each time to run this cell.

In [5]:
```
# import the chain tool to combine lists
from itertools import chain

def stop_reached(centroids_old,\
                 centroids_new,thresh=0.5):
    """a function that compares two lists of
    centroids to determine if coordinate has
    moved a greater distance than the
    threshold, by default set to 0.5"""

    # convert the lists of centroids into a
    # single list because we don't care about
    # the context of the coordinates
    centroids_old = list(chain(*centroids_old))
    centroids_new = list(chain(*centroids_new))

    # calculate the difference between each
    # of the coordinates
    difference = [abs(old-new) for old,new in\
                  zip(centroids_old,centroids_new)]

    # set the flag for stopping to true
    # by default
    stopping = True

    # loop through each difference
    for diff in difference:

        # if the difference is greater
        # than the threshold, then break
        # out of the loop and set the
        # indicator for stopping to
        # false
        if diff > thresh:
            stopping = False
            break

    # return whether or not we reached the
    # threshold or we need to keep going
    return stopping
```

Use a runner to run the MRJob class in the notebook

```
# import the MRJob that we created
from mr_kmeans import MRKmeans

# import pandas to help us save and load
# the centroids
import pandas as pd

# set the data that we're going to pull
mr_job = MRKmeans(args=['twitter_users_norm.txt','--
file=centroids.txt'])

# read in the centroids data to get the original
# centroids and convert it to a list
centroids = pd.read_csv('centroids.txt',\
                        header=None)
centroids = map(list,centroids.values)

# create a counter to count our iterations
# and an initial stopping indicator
iteration = 0
stopping = False

# set up a loop that runs until we tell
# it to stop
while stopping == False:

    # set the old centroids
    old_centroids = centroids[:]

    # create a new array to hold the
    # new centroid points
    new_centroids = []

    # print the iteration we are on
    print "\n*~*~*~*~*~*~*~*\n"
    print "Iteration:", iteration

    # create the runner and run it
    with mr_job.make_runner() as runner:
        runner.run()

        # stream_output: get access of the output
        for line in runner.stream_output():

            # set the centroid
            index,coordinates = mr_job.parse_output_line(line)

            # update the current centroid
            new_centroids.append(coordinates)

            # print out the centroid values
            print "Index:", index
            print "Coordinates sample:", coordinates[0:4]

        # set the new centroids as a regular list
        new_centroids = new_centroids[:]
        centroids = new_centroids[:]
```

```
    # convert our array to a pandas data frame
    # and write that data frame to an output file
    # and update the centroids file
    centroids_pd = pd.DataFrame(centroids)
    centroids_pd.to_csv('HW4.5_B_Output',\
                     header=False,index=False)
    centroids_pd.to_csv('centroids.txt',\
                     header=False,index=False)

    # check the stopping condition with our
    # new and old centroids
    stopping = stop_reached(old_centroids,\
                            new_centroids,thresh=0.001)

    # iterate the iteration count by 1
    iteration = iteration + 1
```

```
*~*~*~*~*~*~*~*

Iteration: 0
Cluster # 0
        Class 0         0.803191489362
        Class 1         0.582417582418
        Class 2         0.185185185185
        Class 3         0.388349514563
Cluster # 1
        Class 0         0.196808510638
        Class 1         0.417582417582
        Class 2         0.814814814815
        Class 3         0.611650485437
Index: 0
Coordinates sample: [0.019651753250353256, 0.04433178480303183, 0.02460
2159831210973, 0.025342965897744065]
Index: 1
Coordinates sample: [0.054432759077732026, 0.034418995893678764, 0.0225
1726979696211, 0.022504075700255012]

*~*~*~*~*~*~*~*

Iteration: 1
Cluster # 0
        Class 0         0.974734042553
        Class 1         0.021978021978
        Class 2         0.12962962963
        Class 3         0.621359223301
Cluster # 1
        Class 0         0.0252659574468
        Class 1         0.978021978022
        Class 2         0.87037037037
        Class 3         0.378640776699
Index: 0
Coordinates sample: [0.012301863884069024, 0.04763117086527928, 0.02519
2921102346992, 0.02690762951496705]
Index: 1
Coordinates sample: [0.10271796735678142, 0.0156526491503931, 0.0189989
31147857532, 0.01455475093204292]

*~*~*~*~*~*~*~*

Iteration: 2
Cluster # 0
        Class 0         0.998670212766
        Class 1         0.032967032967
        Class 2         0.240740740741
        Class 3         0.922330097087
Cluster # 1
        Class 0         0.00132978723404
        Class 1         0.967032967033
        Class 2         0.759259259259
        Class 3         0.0776699029126
Index: 0
Coordinates sample: [0.014353908136708856, 0.047148667719523356, 0.0247
21456271876878, 0.0265098310180887]
Index: 1
```

```
Coordinates sample: [0.1265907184342913, 0.005689754191030542, 0.019430
374962450526, 0.012026787915125448]

*~*~*~*~*~*~*~*

Iteration: 3
Cluster # 0
        Class 0         0.998670212766
        Class 1         0.032967032967
        Class 2         0.259259259259
        Class 3         0.961165048544
Cluster # 1
        Class 0         0.00132978723404
        Class 1         0.967032967033
        Class 2         0.740740740741
        Class 3         0.0388349514563
Index: 0
Coordinates sample: [0.014694596354347216, 0.04700762291410899, 0.02475
6628910398697, 0.026451644460645893]
Index: 1
Coordinates sample: [0.12858927006433232, 0.005050590872622954, 0.01900
2178841055435, 0.011861618966163773]

*~*~*~*~*~*~*~*

Iteration: 4
Cluster # 0
        Class 0         0.998670212766
        Class 1         0.032967032967
        Class 2         0.259259259259
        Class 3         0.961165048544
Cluster # 1
        Class 0         0.00132978723404
        Class 1         0.967032967033
        Class 2         0.740740740741
        Class 3         0.0388349514563
Index: 0
Coordinates sample: [0.014694596354347216, 0.04700762291410899, 0.02475
6628910398697, 0.026451644460645893]
Index: 1
Coordinates sample: [0.12858927006433232, 0.005050590872622954, 0.01900
2178841055435, 0.011861618966163773]
```

Part C. K=4 Petrubation centroids from the aggregated data

Generate the random centroids

In [2]:
```
# import pandas to help us write to csvs
import pandas as pd

# generate 4 centroids from the aggregate
# data
centroids = aggregateCentroids(4)

# write the data to a centroids file
centroids_pd = pd.DataFrame(centroids)
centroids_pd.to_csv('centroids.txt',\
                    header=False,index=False)

# read the first couple lines
print "Done. We got", len(centroids), "centroids"
```

```
Done. We got 4 centroids
```

Write the MRJob class to find the best centroids

```
%%writefile mr_kmeans.py
# import MRJob and some other libraries
# to help us get started
from mrjob.job import MRJob
from mrjob.step import MRStep
import numpy as np
import re
import pandas as pd

# define a function that will find which centroid
# is closest to a given point
def ClosestCentroid(point,centroid_points):
    """takes a point, a list of coordinates, and
    compares that point to each of a number of
    centroids stored in a list of lists. returns
    the index of the centroid closest to the data
    point"""

    # convert our inputs into numpy arrays
    point = np.array(point)
    centroid_points = np.array(centroid_points)

    # calculate the difference between the point
    # and each of the centroid points
    difference = point - centroid_points

    # square the difference, this will help us
    # calculate distance regardless of direction
    diff_sq = difference * difference

    # get the index of the centroid that is
    # closest to the data point
    closest_index = \
    np.argmin(list(diff_sq.sum(axis=1)))

    # return the closest index
    return int(closest_index)


# create the MRJob class
class MRKmeans(MRJob):
    """class responsible for find the nearest centroid
    to a number of data points"""

    # create an array to hold our centroid
    # points and set a value for K, number
    # of centroids
    centroid_points = []
    K=4

    # set the number of true classifications
    # and set the number of dimensions in our
    # data
    TRUTHS = 4
    DIMS = 1000

    # read in the class count file
```

```
    classes_pd = pd.read_csv('class_counts.txt',\
                             header=None)

    # set the class counts
    class_counts = map(float,classes_pd.values)

    # create an empty array to hold the counts
    # for each class
    classes = [0] * TRUTHS

    # define the steps of the job and the order in
    # which they will be executed
    def steps(self):
        return [MRStep(mapper_init=self.mapper_init,\
                       mapper=self.mapper,\
                       combiner=self.combiner,\
                       reducer=self.reducer)]


    # load the initial centroids from a
    # data file passed in
    def mapper_init(self):

        # read in the centroids data
        centroids = pd.read_csv('Centroids.txt',\
                                header=None)

        # set the centroid points based on the
        # inputted file
        self.centroid_points = map(list,centroids.values)


    # takes a line of the twitter data and
    # returns the index of the closest centroid
    # and the coordinates of this point,
    # along with this point's true class
    def mapper(self, _, line):

        # get all the information for the point
        point = map(float,line.split(','))

        # get the point's true classification
        # and simplify the point to just it's
        # coordinates
        truth = int(point[1])
        point = point[3:]

        # grab the closest centroid
        closest = \
        ClosestCentroid(point,self.centroid_points)

        # create an array of zeros of the
        # length of the true classifications
        classify = [0] * self.TRUTHS

        # set the index of the truth to be 1
        classify[truth] = 1
```

```
        # yield:
        # key: the index of the closest cluster
        # value: the coordinates of the point &
        # the classification
        yield closest, (point, classify)


    # takes the output of the mapper and combines
    # the coordinate positions and updates the
    # count of points for this centroid
    def combiner(self, centroid, point_classify):

        # get the centroid value
        centroid = int(centroid)

        # set two blank arrays to hold the sums of
        # the coordinates and the sums of the true
        # classifications
        coordinates = [0] * self.DIMS
        truths = [0] * self.TRUTHS

        # convert our arrays to numpy arrays
        coordinates = np.array(coordinates)
        truths = np.array(truths)

        # loop through each point and its
        # associated classification
        for point,classify in point_classify:

            # set each element as a numpy array
            point = np.array(point)
            classify = np.array(classify)

            # sum the coordinates and
            # classification values
            coordinates = coordinates + point
            truths = truths + classify

        # convert the numpy arrays back to
        # regular arrays for the combiner's
        # output
        coordinates = list(coordinates)
        truths = list(truths)

        # yield the key as the centroid and the
        # sum of the coordinates and the sum of
        # the classifications
        yield centroid, (coordinates,truths)

    # takes the outputs of the mappers and
    # combiners and computes the aggregate
    # sums for each centroid and uses these
    # sums to calculate new centroids at
    # the centers of the clusters
    def reducer(self, centroid, point_classify):
```

```
        # get the centroid value
        centroid = int(centroid)

        # set two blank arrays to hold the sums of
        # the coordinates and the sums of the true
        # classifications
        coordinates = [0] * self.DIMS
        truths = [0] * self.TRUTHS

        # convert our arrays to numpy arrays
        coordinates = np.array(coordinates)
        truths = np.array(truths)

        # loop through each point and its
        # associated classification
        for point,classify in point_classify:

            # set each element as a numpy array
            point = np.array(point)
            classify = np.array(classify)

            # sum the coordinates and
            # classification values
            coordinates = coordinates + point
            truths = truths + classify

        # gather the complete count for the
        # centroid
        num_points = float(sum(truths))

        # calculate the new centroid and
        # convert it back to a regular list
        new_centroid = coordinates / num_points
        new_centroid = list(new_centroid)

        # print out the class breakdown
        print "Cluster #",centroid
        for index,item in enumerate(truths):
            proportion = float(item) /\
            self.class_counts[index]

            print "\tClass",index,"\t",proportion

        # yield the centroid index and the
        # coordinates of the new centroid
        yield centroid, new_centroid
```

```
Overwriting mr_kmeans.py
```

Create a copy of the stop function to tell us when we have achieved sufficient convergence

We put a copy down here because we don't want to have to scroll each time to run this cell.

In [4]:
```
# import the chain tool to combine lists
from itertools import chain

def stop_reached(centroids_old,\
                 centroids_new,thresh=0.5):
    """a function that compares two lists of
    centroids to determine if coordinate has
    moved a greater distance than the
    threshold, by default set to 0.5"""

    # convert the lists of centroids into a
    # single list because we don't care about
    # the context of the coordinates
    centroids_old = list(chain(*centroids_old))
    centroids_new = list(chain(*centroids_new))

    # calculate the difference between each
    # of the coordinates
    difference = [abs(old-new) for old,new in\
                 zip(centroids_old,centroids_new)]

    # set the flag for stopping to true
    # by default
    stopping = True

    # loop through each difference
    for diff in difference:

        # if the difference is greater
        # than the threshold, then break
        # out of the loop and set the
        # indicator for stopping to
        # false
        if diff > thresh:
            stopping = False
            break

    # return whether or not we reached the
    # threshold or we need to keep going
    return stopping
```

Use a runner to run the MRJob class in the notebook

```
# import the MRJob that we created
from mr_kmeans import MRKmeans

# import pandas to help us save and load
# the centroids
import pandas as pd

# set the data that we're going to pull
mr_job = MRKmeans(args=['twitter_users_norm.txt','--
file=centroids.txt'])

# read in the centroids data to get the original
# centroids and convert it to a list
centroids = pd.read_csv('centroids.txt',\
                        header=None)
centroids = map(list,centroids.values)

# create a counter to count our iterations
# and an initial stopping indicator
iteration = 0
stopping = False

# set up a loop that runs until we tell
# it to stop
while stopping == False:

    # set the old centroids
    old_centroids = centroids[:]

    # create a new array to hold the
    # new centroid points
    new_centroids = []

    # print the iteration we are on
    print "\n*~*~*~*~*~*~*~*~*\n"
    print "Iteration:", iteration

    # create the runner and run it
    with mr_job.make_runner() as runner:
        runner.run()

        # stream_output: get access of the output
        for line in runner.stream_output():

            # set the centroid
            index,coordinates = mr_job.parse_output_line(line)

            # update the current centroid
            new_centroids.append(coordinates)

            # print out the centroid values
            print "Index:", index
            print "Coordinates sample:", coordinates[0:4]

        # set the new centroids as a regular list
        new_centroids = new_centroids[:]
        centroids = new_centroids[:]
```

```
        # convert our array to a pandas data frame
        # and write that data frame to an output file
        # and update the centroids file
        centroids_pd = pd.DataFrame(centroids)
        centroids_pd.to_csv('HW4.5_C_Output',\
                        header=False,index=False)
        centroids_pd.to_csv('centroids.txt',\
                        header=False,index=False)

        # check the stopping condition with our
        # new and old centroids
        stopping = stop_reached(old_centroids,\
                               new_centroids,thresh=0.001)

        # iterate the iteration count by 1
        iteration = iteration + 1
```

```
*~*~*~*~*~*~*~*

Iteration: 0
Cluster # 0
        Class 0         0.0252659574468
        Class 1         0.10989010989
        Class 2         0.314814814815
        Class 3         0.0388349514563
Cluster # 1
        Class 0         0.31914893617
        Class 1         0.461538461538
        Class 2         0.37037037037
        Class 3         0.0970873786408
Cluster # 2
        Class 0         0.610372340426
        Class 1         0.010989010989
        Class 2         0.148148148148
        Class 3         0.631067961165
Cluster # 3
        Class 0         0.0452127659574
        Class 1         0.417582417582
        Class 2         0.166666666667
        Class 3         0.233009708738
Index: 0
Coordinates sample: [0.05911350902286960 4, 0.013895849784602338, 0.0277
13639159403936, 0.012885151120974783]
Index: 1
Coordinates sample: [0.031376541404044056, 0.033644801373359755, 0.0263
63983237804867, 0.02227330736146639]
Index: 2
Coordinates sample: [0.014840876675482475, 0.05195407622060502, 0.02251
242086039605, 0.027094200815265014]
Index: 3
Coordinates sample: [0.08749756495750337, 0.024227090564671958, 0.02267
5447661139495, 0.023585072214450022]

*~*~*~*~*~*~*~*

Iteration: 1
Cluster # 0
        Class 0         0.0
        Class 1         0.538461538462
        Class 2         0.5
        Class 3         0.0
Cluster # 1
        Class 0         0.0984042553191
        Class 1         0.021978021978
        Class 2         0.111111111111
        Class 3         0.0679611650485
Cluster # 2
        Class 0         0.897606382979
        Class 1         0.010989010989
        Class 2         0.0185185185185
        Class 3         0.78640776699
Cluster # 3
        Class 0         0.00398936170213
        Class 1         0.428571428571
```

```
        Class 2         0.37037037037
        Class 3         0.145631067961
Index: 0
Coordinates sample: [0.09345373621417893, 0.004040559200611162, 0.01144
7188819297294, 0.0021880530640920382]
Index: 1
Coordinates sample: [0.02308316910131965, 0.02476676950214827, 0.034823
30086368247, 0.02476991001744669]
Index: 2
Coordinates sample: [0.012795635043544444, 0.05020126555698912, 0.02379
1936377802397, 0.026974152054583797]
Index: 3
Coordinates sample: [0.14268325444770782, 0.01121368020072684, 0.025814
79415152595, 0.02199934789716243]

*~*~*~*~*~*~*~*

Iteration: 2
Cluster # 0
        Class 0         0.0
        Class 1         0.56043956044
        Class 2         0.037037037037
        Class 3         0.0
Cluster # 1
        Class 0         0.18085106383
        Class 1         0.021978021978
        Class 2         0.259259259259
        Class 3         0.184466019417
Cluster # 2
        Class 0         0.817819148936
        Class 1         0.010989010989
        Class 2         0.0
        Class 3         0.766990291262
Cluster # 3
        Class 0         0.00132978723404
        Class 1         0.406593406593
        Class 2         0.703703703704
        Class 3         0.0485436893204
Index: 0
Coordinates sample: [0.10519387900156982, 0.0021734038420666737, 0.0002
1414346101234543, 0.0004958214585589612]
Index: 1
Coordinates sample: [0.018340617328336556, 0.02298504918701455, 0.03196
4945169413735, 0.02575739184247389]
Index: 2
Coordinates sample: [0.01368041163398271, 0.05292044918632553, 0.023002
47037610911, 0.02664973941357403]
Index: 3
Coordinates sample: [0.1434960582954136, 0.007431995722050855, 0.031200
18410341751, 0.019244563396038975]

*~*~*~*~*~*~*~*

Iteration: 3
Cluster # 0
        Class 0         0.0
        Class 1         0.56043956044
```

```
        Class 2         0.037037037037
        Class 3         0.0
Cluster # 1
        Class 0         0.191489361702
        Class 1         0.021978021978
        Class 2         0.259259259259
        Class 3         0.223300970874
Cluster # 2
        Class 0         0.807180851064
        Class 1         0.010989010989
        Class 2         0.0
        Class 3         0.73786407767
Cluster # 3
        Class 0         0.00132978723404
        Class 1         0.406593406593
        Class 2         0.703703703704
        Class 3         0.0388349514563
Index: 0
Coordinates sample: [0.10519387900156982, 0.0021734038420666737, 0.0002
1414346101234543, 0.0004958214585589612]
Index: 1
Coordinates sample: [0.016697865673743068, 0.022807412658713604, 0.0310
03003582612893, 0.026057495946701828]
Index: 2
Coordinates sample: [0.014158633948719376, 0.053482240570157824, 0.0230
85449721780008, 0.026557096475341452]
Index: 3
Coordinates sample: [0.14408871664341252, 0.00695672728036649, 0.031449
25228033398, 0.01939145981495196]

*~*~*~*~*~*~*~*

Iteration: 4
Cluster # 0
        Class 0         0.0
        Class 1         0.56043956044
        Class 2         0.037037037037
        Class 3         0.0
Cluster # 1
        Class 0         0.186170212766
        Class 1         0.021978021978
        Class 2         0.259259259259
        Class 3         0.233009708738
Cluster # 2
        Class 0         0.8125
        Class 1         0.010989010989
        Class 2         0.0
        Class 3         0.728155339806
Cluster # 3
        Class 0         0.00132978723404
        Class 1         0.406593406593
        Class 2         0.703703703704
        Class 3         0.0388349514563
Index: 0
Coordinates sample: [0.10519387900156982, 0.0021734038420666737, 0.0002
1414346101234543, 0.0004958214585589612]
Index: 1
```

```
Coordinates sample: [0.015853401419623457, 0.022236491341875716, 0.0308
31595049569054, 0.025806544490375744]
Index: 2
Coordinates sample: [0.014390979306676582, 0.05349787572779463, 0.02316
4934725463254, 0.026620666286917543]
Index: 3
Coordinates sample: [0.14408871664341252, 0.00695672728036649, 0.031449
25228033398, 0.01939145981495196]
```

Part D. K=4 trained centroids

Use the row-level normalized aggregates as 'trained' centroids.

Write a function calculate the row-level normalized aggregates trained centroids by class

In [1]:
```
# import the numpy library to help us
# with randomization
import numpy as np
import re

def trainedCentroids(k=4):
    """generates k centroid points from the
    aggregate data that is trained by
    normalizing the aggregate data for each
    class"""

    # initalize a counter at zero
    counter = 0

    # create an array to hold the coordinates for
    # each centroid
    centroids = []

    # loop through each line in the summary data
    for line in \
    open("topUsers_Apr-Jul_2014_1000-words_summaries.txt")\
    .readlines():

        # if it's between the 3rd and 6th lines
        if counter >= 2 and counter <= 5:

            # split the line by commas
            data = re.split(",",line)

            # calculate the class aggregate as
            # the normalized count for each word
            classAggregate = \
            [float(data[i+3])/float(data[2]) \
             for i in range(1000)]

            # add our class aggregate to the
            # to the centroids list
            centroids.append(classAggregate)

        # increment our line counter by 1
        counter += 1

    # return the new centroids
    return centroids
```

Generate the 'trained' centroids

In [2]:
```
# import pandas to help us write to csvs
import pandas as pd

# generate 4 trained centroids from the
# aggregate data
centroids = trainedCentroids(4)

# write the data to a centroids file
centroids_pd = pd.DataFrame(centroids)
centroids_pd.to_csv('centroids.txt',\
                    header=False,index=False)

# read the first couple lines
print "Done. We got", len(centroids), "trained centroids"
```

```
Done. We got 4 trained centroids
```

Write the MRJob class to find the best centroids

```
%%writefile mr_kmeans.py
# import MRJob and some other libraries
# to help us get started
from mrjob.job import MRJob
from mrjob.step import MRStep
import numpy as np
import re
import pandas as pd

# define a function that will find which centroid
# is closest to a given point
def ClosestCentroid(point,centroid_points):
    """takes a point, a list of coordinates, and
    compares that point to each of a number of
    centroids stored in a list of lists. returns
    the index of the centroid closest to the data
    point"""

    # convert our inputs into numpy arrays
    point = np.array(point)
    centroid_points = np.array(centroid_points)

    # calculate the difference between the point
    # and each of the centroid points
    difference = point - centroid_points

    # square the difference, this will help us
    # calculate distance regardless of direction
    diff_sq = difference * difference

    # get the index of the centroid that is
    # closest to the data point
    closest_index = \
    np.argmin(list(diff_sq.sum(axis=1)))

    # return the closest index
    return int(closest_index)


# create the MRJob class
class MRKmeans(MRJob):
    """class responsible for find the nearest centroid
    to a number of data points"""

    # create an array to hold our centroid
    # points and set a value for K, number
    # of centroids
    centroid_points = []
    K=4

    # set the number of true classifications
    # and set the number of dimensions in our
    # data
    TRUTHS = 4
    DIMS = 1000

    # read in the class count file
```

```
    classes_pd = pd.read_csv('class_counts.txt',\
                         header=None)

    # set the class counts
    class_counts = map(float,classes_pd.values)

    # create an empty array to hold the counts
    # for each class
    classes = [0] * TRUTHS

    # define the steps of the job and the order in
    # which they will be executed
    def steps(self):
        return [MRStep(mapper_init=self.mapper_init,\
                       mapper=self.mapper,\
                       combiner=self.combiner,\
                       reducer=self.reducer)]


    # load the initial centroids from a
    # data file passed in
    def mapper_init(self):

        # read in the centroids data
        centroids = pd.read_csv('Centroids.txt',\
                                header=None)

        # set the centroid points based on the
        # inputted file
        self.centroid_points = map(list,centroids.values)


    # takes a line of the twitter data and
    # returns the index of the closest centroid
    # and the coordinates of this point,
    # along with this point's true class
    def mapper(self, _, line):

        # get all the information for the point
        point = map(float,line.split(','))

        # get the point's true classification
        # and simplify the point to just it's
        # coordinates
        truth = int(point[1])
        point = point[3:]

        # grab the closest centroid
        closest = \
        ClosestCentroid(point,self.centroid_points)

        # create an array of zeros of the
        # length of the true classifications
        classify = [0] * self.TRUTHS

        # set the index of the truth to be 1
        classify[truth] = 1
```

```
        # yield:
        # key: the index of the closest cluster
        # value: the coordinates of the point &
        # the classification
        yield closest, (point, classify)


    # takes the output of the mapper and combines
    # the coordinate positions and updates the
    # count of points for this centroid
    def combiner(self, centroid, point_classify):

        # get the centroid value
        centroid = int(centroid)

        # set two blank arrays to hold the sums of
        # the coordinates and the sums of the true
        # classifications
        coordinates = [0] * self.DIMS
        truths = [0] * self.TRUTHS

        # convert our arrays to numpy arrays
        coordinates = np.array(coordinates)
        truths = np.array(truths)

        # loop through each point and its
        # associated classification
        for point,classify in point_classify:

            # set each element as a numpy array
            point = np.array(point)
            classify = np.array(classify)

            # sum the coordinates and
            # classification values
            coordinates = coordinates + point
            truths = truths + classify

        # convert the numpy arrays back to
        # regular arrays for the combiner's
        # output
        coordinates = list(coordinates)
        truths = list(truths)

        # yield the key as the centroid and the
        # sum of the coordinates and the sum of
        # the classifications
        yield centroid, (coordinates,truths)

    # takes the outputs of the mappers and
    # combiners and computes the aggregate
    # sums for each centroid and uses these
    # sums to calculate new centroids at
    # the centers of the clusters
    def reducer(self, centroid, point_classify):
```

```
        # get the centroid value
        centroid = int(centroid)

        # set two blank arrays to hold the sums of
        # the coordinates and the sums of the true
        # classifications
        coordinates = [0] * self.DIMS
        truths = [0] * self.TRUTHS

        # convert our arrays to numpy arrays
        coordinates = np.array(coordinates)
        truths = np.array(truths)

        # loop through each point and its
        # associated classification
        for point,classify in point_classify:

            # set each element as a numpy array
            point = np.array(point)
            classify = np.array(classify)

            # sum the coordinates and
            # classification values
            coordinates = coordinates + point
            truths = truths + classify

        # gather the complete count for the
        # centroid
        num_points = float(sum(truths))

        # calculate the new centroid and
        # convert it back to a regular list
        new_centroid = coordinates / num_points
        new_centroid = list(new_centroid)

        # print out the class breakdown
        print "Cluster #",centroid
        for index,item in enumerate(truths):
            proportion = float(item) /\
            self.class_counts[index]

            print "\tClass",index,"\t",proportion

        # yield the centroid index and the
        # coordinates of the new centroid
        yield centroid, new_centroid
```

```
Overwriting mr_kmeans.py
```

Create a copy of the stop function to tell us when we have achieved sufficient convergence

We put a copy down here because we don't want to have to scroll each time to run this cell.

In [4]:
```
# import the chain tool to combine lists
from itertools import chain

def stop_reached(centroids_old,\
                 centroids_new,thresh=0.5):
    """a function that compares two lists of
    centroids to determine if coordinate has
    moved a greater distance than the
    threshold, by default set to 0.5"""

    # convert the lists of centroids into a
    # single list because we don't care about
    # the context of the coordinates
    centroids_old = list(chain(*centroids_old))
    centroids_new = list(chain(*centroids_new))

    # calculate the difference between each
    # of the coordinates
    difference = [abs(old-new) for old,new in\
                  zip(centroids_old,centroids_new)]

    # set the flag for stopping to true
    # by default
    stopping = True

    # loop through each difference
    for diff in difference:

        # if the difference is greater
        # than the threshold, then break
        # out of the loop and set the
        # indicator for stopping to
        # false
        if diff > thresh:
            stopping = False
            break

    # return whether or not we reached the
    # threshold or we need to keep going
    return stopping
```

Use a runner to run the MRJob class in the notebook

```
# import the MRJob that we created
from mr_kmeans import MRKmeans

# import pandas to help us save and load
# the centroids
import pandas as pd

# set the data that we're going to pull
mr_job = MRKmeans(args=['twitter_users_norm.txt','--
file=centroids.txt'])

# read in the centroids data to get the original
# centroids and convert it to a list
centroids = pd.read_csv('centroids.txt',\
                        header=None)
centroids = map(list,centroids.values)

# create a counter to count our iterations
# and an initial stopping indicator
iteration = 0
stopping = False

# set up a loop that runs until we tell
# it to stop
while stopping == False:

    # set the old centroids
    old_centroids = centroids[:]

    # create a new array to hold the
    # new centroid points
    new_centroids = []

    # print the iteration we are on
    print "\n*~*~*~*~*~*~*~*\n"
    print "Iteration:", iteration

    # create the runner and run it
    with mr_job.make_runner() as runner:
        runner.run()

        # stream_output: get access of the output
        for line in runner.stream_output():

            # set the centroid
            index,coordinates = mr_job.parse_output_line(line)

            # update the current centroid
            new_centroids.append(coordinates)

            # print out the centroid values
            print "Index:", index
            print "Coordinates sample:", coordinates[0:4]

        # set the new centroids as a regular list
        new_centroids = new_centroids[:]
        centroids = new_centroids[:]
```

```
        # convert our array to a pandas data frame
        # and write that data frame to an output file
        # and update the centroids file
        centroids_pd = pd.DataFrame(centroids)
        centroids_pd.to_csv('HW4.5_D_Output',\
                        header=False,index=False)
        centroids_pd.to_csv('centroids.txt',\
                        header=False,index=False)

        # check the stopping condition with our
        # new and old centroids
        stopping = stop_reached(old_centroids,\
                               new_centroids,thresh=0.001)

        # iterate the iteration count by 1
        iteration = iteration + 1
```

```
*~*~*~*~*~*~*~*

Iteration: 0
Cluster # 0
        Class 0         0.996010638298
        Class 1         0.032967032967
        Class 2         0.0555555555556
        Class 3         0.31067961165
Cluster # 1
        Class 0         0.0
        Class 1         0.637362637363
        Class 2         0.0555555555556
        Class 3         0.0
Cluster # 2
        Class 0         0.00132978723404
        Class 1         0.32967032967
        Class 2         0.888888888889
        Class 3         0.0291262135922
Cluster # 3
        Class 0         0.00265957446809
        Class 1         0.0
        Class 2         0.0
        Class 3         0.660194174757
Index: 0
Coordinates sample: [0.013056587935590983, 0.047863000573448035, 0.0258
128158139248, 0.027212560757059225]
Index: 1
Coordinates sample: [0.13757453449382295, 0.0025576324607467582, 0.0020
380910562584967, 0.003116342125949376]
Index: 2
Coordinates sample: [0.1054368093820366, 0.005971592697524741, 0.029885
185038034556, 0.01689746846828232]
Index: 3
Coordinates sample: [0.03613126112878477, 0.04447814314264557, 0.015738
461115237426, 0.021702806714172827]

*~*~*~*~*~*~*~*

Iteration: 1
Cluster # 0
        Class 0         0.996010638298
        Class 1         0.032967032967
        Class 2         0.148148148148
        Class 3         0.349514563107
Cluster # 1
        Class 0         0.0
        Class 1         0.571428571429
        Class 2         0.037037037037
        Class 3         0.0
Cluster # 2
        Class 0         0.00132978723404
        Class 1         0.395604395604
        Class 2         0.814814814815
        Class 3         0.0388349514563
Cluster # 3
        Class 0         0.00265957446809
        Class 1         0.0
```

```
        Class 2         0.0
        Class 3         0.611650485437
Index: 0
Coordinates sample: [0.01302441329089856, 0.04747984207751542, 0.025666
62757573825, 0.027057683816590336]
Index: 1
Coordinates sample: [0.11975834018130002, 0.002291815024006652, 0.00021
66901341305983, 0.0005012922384216841]
Index: 2
Coordinates sample: [0.12534419729754473, 0.006273960793754537, 0.02997
955000006059, 0.018509928285996367]
Index: 3
Coordinates sample: [0.03621458212520745, 0.045789825848055375, 0.01539
5207440618307, 0.021120631795221504]

*~*~*~*~*~*~*~*

Iteration: 2
Cluster # 0
        Class 0         0.996010638298
        Class 1         0.032967032967
        Class 2         0.222222222222
        Class 3         0.368932038835
Cluster # 1
        Class 0         0.0
        Class 1         0.56043956044
        Class 2         0.0
        Class 3         0.0
Cluster # 2
        Class 0         0.00132978723404
        Class 1         0.406593406593
        Class 2         0.777777777778
        Class 3         0.0388349514563
Cluster # 3
        Class 0         0.00265957446809
        Class 1         0.0
        Class 2         0.0
        Class 3         0.592233009709
Index: 0
Coordinates sample: [0.0130970327448467, 0.047262693074177284, 0.025535
56020957466, 0.026916487721991778]
Index: 1
Coordinates sample: [0.10931912915849412, 0.0022586353652849747, 0.0002
225412437971433, 0.0005152654373259792]
Index: 2
Coordinates sample: [0.13774887556667859, 0.006631163234410254, 0.03034
929542949903, 0.018517260694573653]
Index: 3
Coordinates sample: [0.0348029612972061, 0.045245233202876184, 0.015096
46577119984, 0.02130824573324679]

*~*~*~*~*~*~*~*

Iteration: 3
Cluster # 0
        Class 0         0.996010638298
        Class 1         0.032967032967
```

```
        Class 2         0.259259259259
        Class 3         0.368932038835
Cluster # 1
        Class 0         0.0
        Class 1         0.56043956044
        Class 2         0.0
        Class 3         0.0
Cluster # 2
        Class 0         0.00132978723404
        Class 1         0.406593406593
        Class 2         0.740740740741
        Class 3         0.0388349514563
Cluster # 3
        Class 0         0.00265957446809
        Class 1         0.0
        Class 2         0.0
        Class 3         0.592233009709
Index: 0
Coordinates sample: [0.013118940892406785, 0.04714572061536234, 0.02551
358199220159, 0.026854671972867464]
Index: 1
Coordinates sample: [0.10931912915849412, 0.0022586353652849747, 0.0002
225412437971433, 0.0005152654373259792]
Index: 2
Coordinates sample: [0.14057435770089027, 0.0067870510052356, 0.0306821
973466673, 0.01891849738044094]
Index: 3
Coordinates sample: [0.0348029612972061, 0.045245233202876184, 0.015096
46577119984, 0.02130824573324679]

*~*~*~*~*~*~*~*

Iteration: 4
Cluster # 0
        Class 0         0.996010638298
        Class 1         0.032967032967
        Class 2         0.259259259259
        Class 3         0.368932038835
Cluster # 1
        Class 0         0.0
        Class 1         0.56043956044
        Class 2         0.0
        Class 3         0.0
Cluster # 2
        Class 0         0.00132978723404
        Class 1         0.406593406593
        Class 2         0.740740740741
        Class 3         0.0388349514563
Cluster # 3
        Class 0         0.00265957446809
        Class 1         0.0
        Class 2         0.0
        Class 3         0.592233009709
Index: 0
Coordinates sample: [0.013118940892406785, 0.04714572061536234, 0.02551
358199220159, 0.026854671972867464]
Index: 1
```

```
Coordinates sample: [0.10931912915849412, 0.0022586353652849747, 0.0002
225412437971433, 0.0005152654373259792]
Index: 2
Coordinates sample: [0.14057435770089027, 0.0067870510052356, 0.0306821
973466673, 0.01891849738044094]
Index: 3
Coordinates sample: [0.0348029612972061, 0.045245233202876184, 0.015096
46577119984, 0.02130824573324679]
```

As we look through parts A-D, we notice a marked improvement as we move towards less random, and more trained, clusters. For example, part A which is completely took 9 iterations one time I ran it. On the other hand, part D only took 4 iterations. This is because in part D, we are moving towards the true centers for each class. As we move towards the centers, we have to make less iterations to find the true centers.