Name: Patrick Ng
Email: patng@ischool.berkeley.edu
Class: W261-2
Midterm
Date of submission: Mar 2, 2016

In [11]:
```
%load_ext autoreload
%autoreload 2
```

In [49]:
```
%%writefile kltext.txt
1.Data Science is an interdisciplinary field about processes and systems to extrac
2.Machine learning is a subfield of computer science[1] that evolved from the stud
```

```
Writing kltext.txt
```

MRjob class for calculating pairwise similarity using K-L Divergence as the similarity measure

Job 1: create inverted index (assume just two objects)

Job 2: calculate the similarity of each pair of objects

In [58]:
```
%%writefile kldivergence.py
from mrjob.job import MRJob
import re
import numpy as np
class kldivergence(MRJob):
    def mapper1(self, _, line):
        index = int(line.split('.',1)[0])
        letter_list = re.sub(r"[^A-Za-z]+", '', line).lower()
        count = {}
        for l in letter_list:
            if count.has_key(l):
                count[l] += 1
            else:
                count[l] = 1
        for key in count:
            yield key, [index, count[key]*1.0/len(letter_list)]

    def reducer1(self, key, values):
        counts = [0,0]
        for v in values:
            counts[v[0]-1] += v[1]
        yield key, counts

    def reducer2(self, key, values):
        kl_sum = 0
        for value in values:
            kl_sum = kl_sum + value
        yield None, kl_sum

    def steps(self):
        return [self.mr(mapper=self.mapper1,
                        reducer=self.reducer1),
                self.mr(reducer=self.reducer2)]


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

```
Overwriting kldivergence.py
```

In [ ]:
```
from kldivergence import kldivergence
mr_job = kldivergence(args=['kltext.txt'])
with mr_job.make_runner() as runner:
    runner.run()
    # stream_output: get access of the output
    for line in runner.stream_output():
        print mr_job.parse_output_line(line)
```

Weighted K-means

Write a MapReduce job in MRJob to do the training at scale of a weighted K-means algorithm.

You can write your own code or you can use most of the code from the following notebook:

http://nbviewer.ipython.org/urls/dl.dropbox.com/s/kjtdyi10nwmk4ko/MrJobKmeans-MIDS-Midterm.ipynb
https://www.dropbox.com/s/kjtdyi10nwmk4ko/MrJobKmeans-MIDS-Midterm.ipynb?dl=0

Weight each example as follows using the inverse vector length (Euclidean norm):

weight(X)= 1/||X||,

where ||X|| = SQRT(X.X)= SQRT(X1^2 + X2^2)

Here X is vector made up of X1 and X2.

Using the following data answer the following questions:

https://www.dropbox.com/s/ai1uc3q2ucverly/Kmeandata.csv?dl=0

In [45]:
```
%%writefile Kmeans.py
from __future__ import division
from numpy import argmin, array, random
from mrjob.job import MRJob, MRStep
from itertools import chain
from math import sqrt

def distance(datapoint, centroid_point):
    datapoint = array(datapoint)
    centroid_points = array(centroid_point)
    diff = datapoint - centroid_points
    diffsq = diff**2
    distance = (diffsq.sum(axis = 0))**0.5
    return distance

#Calculate 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**2

    distances = (diffsq.sum(axis = 1))**0.5
    # Get the nearest centroid for each instance
    min_idx = argmin(distances)
    return min_idx

#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=[]
    k=3
    def steps(self):
        return [
            MRStep(mapper_init = self.mapper_init,
                   mapper=self.mapper,
                   #combiner = self.combiner,  # Can't use combiner for MT11
                   reducer_init = self.reducer_init,
                   reducer=self.reducer,
                   reducer_final = self.reducer_final
               )
              ]
    #load centroids info from file
    def mapper_init(self):
        self.centroid_points = [map(float,s.split('\n')[0].split(',')) for s in op
        open('Centroids.txt', 'w').close()

    #load data and output the nearest centroid index and data point
    def mapper(self, _, line):
        D = (map(float,line.split(',')))
        idx = MinDist(D,self.centroid_points)
        # weight(X)= 1/||X||,
        # where ||X|| = SQRT(X.X)= SQRT(X1^2 + X2^2)
        x1 = D[0]
        x2 = D[1]
        weight = sqrt(x1**2 + x2**2)

        yield int(idx), (x1*weight, x2*weight, weight)
```

Driver

Generate random initial centroids

New Centroids = initial centroids

While(1)：

- Cacluate new centroids
- stop if new centroids close to old centroids
- Updates centroids

In [46]:
```
from numpy import random, array
from Kmeans import MRKmeans, stop_criterion
mr_job = MRKmeans(args=['Kmeandata.csv', '--file', 'Centroids.txt',
                        '--strict-protocols', '-r', 'inline'])

#Geneate initial centroids
centroid_points = [[0,0],[6,3],[3,6]]
k = 3
with open('Centroids.txt', 'w+') as f:
        f.writelines(','.join(str(j) for j in i) + '\n' for i in centroid_points)

# Update centroids iteratively
for i in range(10):
    # save previous centoids to check convergency
    centroid_points_old = centroid_points[:]
    print "iteration"+str(i+1)+":"
    with mr_job.make_runner() as runner:
        runner.run()
        # stream_output: get access of the output
        for line in runner.stream_output():
            key,value =  mr_job.parse_output_line(line)
            print key, value
            if key is not None:
                centroid_points[key] = value
    print "\n"
    i = i + 1
print "Centroids\n"
print centroid_points
```

```
iteration1:
0 [-3.9707251767434597, 0.24753995934048853]
1 [5.559358757604786, 0.13140683641026107]
2 [0.21319986473145544, 5.559691555704146]
None 27.0864343044


iteration2:
0 [-5.273661830097599, 0.01778068820189739]
1 [5.315666040265944, -0.0191245246454466]
2 [0.07760590556533625, 5.322298286870532]
None 26.2940010771


iteration3:
0 [-5.29872166540091, -0.006290282704146047]
1 [5.315666040265944, -0.0191245246454466]
2 [0.05740025819123362, 5.3015009631419545]
None 26.2896202234


iteration4:
0 [-5.29872166540091, -0.006290282704146047]
1 [5.315666040265944, -0.0191245246454466]
2 [0.05740025819123362, 5.3015009631419545]
None 26.2896202234


iteration5:
0 [-5.29872166540091, -0.006290282704146047]
1 [5.315666040265944, -0.0191245246454466]
2 [0.05740025819123362, 5.3015009631419545]
None 26.2896202234


iteration6:
0 [-5.29872166540091, -0.006290282704146047]
1 [5.315666040265944, -0.0191245246454466]
2 [0.05740025819123362, 5.3015009631419545]
None 26.2896202234
```

MT11.

Using the result of the previous question, which number below is the closest to the average weighted distance between each example and its assigned (closest) centroid? The average weighted distance is defined as

```
sum over i  (weighted_distance_i)     /  sum over i (weight_i)
```

In [ ]: