HW 4.0

MRJob is a python package for running Hadoop streaming jobs. It assists in producing multistep jobs and submitting them to Hadoop job tracker. The difference between MRJob and MapReduce is that MRJob is a package in python, which is creating Hadoop MapReduce jobs that are run in Java.

MRJob has multiple methods to define the parameters of the Hadoop MapReduce job.

The mapper_init() method is used to define an action to run before the mapper processes any input.

The mapper_final() method is used to define an action to run after the mapper reaches the end of the input.

The combiner_final() method is used to define an action to run after the combiner reaches the end of output.

The reducer_final() method is used to define an action after the reducer reaches the end of input.

HW 4.1

Serialization is the process of turning structured objects into a byte stream for transmission over a network or for writing to persistent storage.

In MRJob or Hadoop, it is done automatically in each task (if it wasn't, we would have to keep running json.loads() and json.dumps()).

The default serialization protocol used by jobs to read input on Python 2 is

In [ ]:
```
class mrjob.protocol.RawValueProtocol
```

HW 4.2

In [1]:
```
from mrjob.job import MRJob
import csv
import re

def file_preprocessor(file_name):
    """Given a sting CSV line, return a list of strings."""
    #for row in csv.reader(file_name):
    with open(file_name, 'r') as f:
        for line in f:
            #print line
            line = re.sub('\n', '', line)
            cells = line.split(',')
            #print cells
            if cells[0] == 'C':
                vis_id = cells[2]

            else:
                if cells[0] == 'V':
                    page_id = cells[1]
                    print 'V', vis_id, 1, 'C', page_id
    #return
```

In [3]:
```
file_preprocessor('anonymous-msweb.data')
```

```
V 10001 1 C 1000
V 10001 1 C 1001
V 10001 1 C 1002
V 10002 1 C 1001
V 10002 1 C 1003
V 10003 1 C 1001
V 10003 1 C 1003
V 10003 1 C 1004
V 10004 1 C 1005
V 10005 1 C 1006
V 10006 1 C 1003
V 10006 1 C 1004
V 10007 1 C 1007
V 10008 1 C 1004
V 10009 1 C 1008
V 10009 1 C 1009
V 10010 1 C 1010
V 10010 1 C 1000
V 10010 1 C 1011
```

HW 4.3

Now, we can find the 5 most frequently visited pages. We can reduce this file to a dictionary, and

let the program count the number of occurrences of each page in the list.

Alternatively, we can just use Challenge2 from the class lecture materials for week 4.

In [26]:
```
%%writefile MostFrequentVisits.py
#!/Users/ninakuklisova/miniconda2/envs/jupi/bin/python

from operator import itemgetter
from collections import defaultdict

def webpage_frequency_counter(preprocessed_data):
    webpage_visits_count = {}
    for line in preprocessed_data:
        words = line.parse()
        key = words[4]
        value = 1
        if key in webpage_visits_count.keys():
            webpage_visits_count[key]+=1

        else:
            webpage_visits_count[key] = value
    return sorted(webpage_visits_count.items(), key= itemgetter(1), rever
```

In [5]:
```
#!python MostFrequentVisits.py -r hadoop anonymous-msweb_converted.data
```

HW4.4

We can similarly find the most frequent visitor of each page:

In [6]:
```
%%writefile MostFrequentVisitors.py
#!/Users/ninakuklisova/miniconda2/envs/jupi/bin/python

from collections import defaultdict


def page_visitor_count (preprocessed_data):
    page_visitor_count = defaultdict(dict)
    for line in preprocessed_data:
        words = line.parse()
        visitor_key = words[1]
        page_key = words[4]
        #key = (visitor_key, page_key)
        if (page_key in webpage_visits_count.keys()) and (visitor_key in
                webpage_visits_count[page_key][visitor_key]+=1

        else:
            webpage_visits_count[key] = value
```

In [7]:
```
#!python MostFrequentVisitors.py -r hadoop anonymous-msweb_converted.data
```

HW 4.5 Clustering Tweet Dataset

In [1]:
```
from numpy import random
numbers = random.sample(1000)
import re
import pylab
%matplotlib inline
import numpy as np
import pandas as pd
```

In [2]:
```
# read in the file and normalize by the 'total' column
raw = pd.read_csv('topUsers_Apr-Jul_2014_1000-words.txt', header = None)

raw.ix[:, 3:] = raw.ix[:, 3:].div(raw.ix[:, 2], 'index')

raw.head()
```

Out[2]:

	0	1	2	3	4	5	6	7	8	9
0	1180025371	2	1724608	0.043808	0.000480	0.033401	0.004133	0.002483	0.026484	0
1	284534859	2	827765	0.120714	0.000000	0.017442	0.034623	0.009022	0.031469	0
2	1602852614	2	987334	0.000000	0.002734	0.000000	0.000000	0.000000	0.000000	0
3	2361533634	2	416584	0.134612	0.000132	0.000007	0.000000	0.000000	0.000000	0
4	485013829	1	530484	0.102629	0.000019	0.000000	0.000000	0.000000	0.000000	0

5 rows × 1003 columns

In [3]:
```
# now, create the centroids file

centroids = raw.ix[:, 3:]

centroids.to_csv('centroids')
```

This generalization of K-means MRJob is based on a class example from week 4 and on the Master Solution.

In [5]:
```
%%writefile Kmeans.py
#!/Users/ninakuklisova/miniconda2/envs/jupi/bin/python


from numpy import argmin, array, random
from mrjob.job import MRJob
from mrjob.step import MRStep
from itertools import chain
import os
import re

#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*diff
    # Get the nearest centroid for each instance
    minidx = argmin(list(diffsq.sum(axis = 1)))
```

```
    return minidx

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

class MRKmeans(MRJob):
    centroid_points=[]
    k=3
    def steps(self):
        return [
            MRStep(mapper_init = self.mapper_init, mapper=self.mapper,com
               ]
    #load centroids info from file
    def mapper_init(self):
        print "Current path:", os.path.dirname(os.path.realpath(__file__)

        self.centroid_points = [map(float,s.split('\n')[0].split(',')) fo
        #open('Centroids.txt', 'w').close()

        print "Centroids: ", self.centroid_points

    #load data and output the nearest centroid index and data point
    def mapper(self, _, datstr):
        total = 0
        data = re.split(',',datstr)
        ID = data[0]
        code = int(data[1])
        users = [ID]
        codes = [0,0,0,0]
        codes[code] = 1
        coords = [float(data[i+3])/float(data[2]) for i in range(1000)]
        for coord in coords:
            total += coord

        minDist = 0
        IDX = -1
        for idx in range(len(self.centroid_points)):
            centroid = self.centroid_points[idx]
            dist = 0
            for ix in range(len(coords)):
                dist += (centroid[ix]-coords[ix])**2
```

```
            dist = dist ** 0.5
            if minDist:
                if dist < minDist:
                    minDist = dist
                    IDX = idx
            else:
                minDist = dist
                IDX = idx
        yield (IDX,[users,1,coords,codes])

    ## combiner takes the mapper output and aggregates (sum) by idx-key
    def combiner(self,IDX,data):
        N = 0
        sumCoords = [0*num for num in range(1000)]
        sumCodes = [0,0,0,0]
        users = []
        for line in data:
            users.extend(line[0])
            N += line[1]
            coords = line[2]
            codes = line[3]
            sumCoords = [sumCoords[i]+coords[i] for i in range(len(sumCoo
            sumCodes = [sumCodes[i]+codes[i] for i in range(len(sumCodes)
        yield (IDX,[users,N,sumCoords,sumCodes])

    ## reducer finishes aggregating all mapper outputs
    ## and then takes the means by idx-key.
    def reducer(self,IDX,data):
        N = 0
        sumCoords = [0*num for num in range(1000)]
        sumCodes = [0,0,0,0]
        users = []
        for line in data:
            users.extend(line[0])
            N += line[1]
            coords = line[2]
            codes = line[3]
            sumCoords = [sumCoords[i]+coords[i] for i in range(len(sumCoo
            sumCodes = [sumCodes[i]+codes[i] for i in range(len(sumCodes)
        centroid = [sumCoords[i]/N for i in range(len(sumCoords))]
        yield (IDX,[users,N,centroid,sumCodes])

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

```
Overwriting Kmeans.py
```

In [6]:
```
import re
```

In [7]:
```
%%writefile kMeans_driver.py
```

```
#!/Users/ninakuklisova/miniconda2/envs/jupi/bin/python


from numpy import random
from Kmeans import MRKmeans
import re,sys

mr_job = MRKmeans(args=["topUsers_Apr-Jul_2014_1000-words.txt","--file","

thresh = 0.0001

scriptName,part = sys.argv

## only stop when distance is below thresh for all centroids
def stopSignal(k,thresh,newCentroids,oldCentroids):
    stop = 1
    for i in range(k):
        dist = 0
        for j in range(len(newCentroids[i])):
            dist += (newCentroids[i][j] - oldCentroids[i][j]) ** 2
        dist = dist ** 0.5
        if (dist > thresh):
            stop  = 0
            break
    return stop

## these are the initialization cases that we want to distinguish:
## A: uniform random centroid-distributions over the 1000 words
def startCentroidsA():
    k = 4
    centroids = []
    for i in range(k):
        rndpoints = random.sample(1000)
        total = sum(rndpoints)
        centroid = [pt/total for pt in rndpoints]
        centroids.append(centroid)
    return centroids

## B: perturbation-centroids, randomly perturbed from the aggregated (use
## C: perturbation-centroids, randomly perturbed from the aggregated (use
## (these were given in the announcement of the exercise)

def startCentroidsBC(k):
    counter = 0
    for line in open("topUsers_Apr-Jul_2014_1000-words_summaries.txt").re
        if counter == 2:
            data = re.split(",",line)
            globalAggregate = [float(data[i+3])/float(data[2]) for i in r
        counter += 1
    centroids = []
    for i in range(k):
```

```
    for i in range(k):
        rndpoints = random.sample(1000)
        peturpoints = [rndpoints[n]/10+globalAggregate[n] for n in range(
        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

## D: "trained" centroids, determined by the sums across the classes, in
def startCentroidsD():
    k = 4
    centroids = []
    counter = 0
    for line in open("topUsers_Apr-Jul_2014_1000-words_summaries.txt").re
        if counter and counter > 1:
            data = re.split(",",line)
            coords = [float(data[i+3])/float(data[2]) for i in range(1000
            centroids.append(coords)
        counter += 1
    return centroids

if part == "A":
    k = 4
    centroids = startCentroidsA()
if part == "B":
    k = 2
    centroids = startCentroidsBC(k)
if part == "C":
    k = 4
    centroids = startCentroidsBC(k)
if part == "D":
    k = 4
    centroids = startCentroidsD()

## the totals for each user type
numType = [752,91,54,103]
numType = [float(numType[i]) for i in range(4)]

with open("centroids.csv", 'w+') as f:
    for centroid in centroids:
        centroid = [str(coord) for coord in centroid]
        f.writelines(",".join(centroid) + "\n")

iterate = 0
stop = 0

clusters = ["NA" for i in range(k)]
N = ["NA" for i in range(k)]
```

```
while(not stop):
    with mr_job.make_runner() as runner:
        runner.run()
        oldCentroids = centroids[:]
        clusterPurities = []
        for line in runner.stream_output():
            key,value =  mr_job.parse_output_line(line)
            clusters[key] = value[0]
            N[key] = value[1]
            centroids[key] = value[2]
            sumCodes = value[3]
            clusterPurities.append(float(max(sumCodes))/float(sum(sumCode

        with open("centroids.csv", 'w+') as f:
            for centroid in centroids:
                centroid = [str(coord) for coord in centroid]
                f.writelines(",".join(centroid) + "\n")

        print str(iterate+1)+","+",".join(str(purity) for purity in clust
        stop = stopSignal(k,thresh,centroids,oldCentroids)
        if not iterate:
            stop = 0
    iterate += 1
```

```
Overwriting kMeans_driver.py
```

In [8]:
```
!chmod +x Kmeans.py kMeans_driver.py
```

In [ ]:
```
#we print the results into separate files
```

In [21]:
```
!./kMeans_driver.py A > purities-A.txt
```

In [130]:
```
!./kMeans_driver.py B > purities-B.txt
```

In [131]:
```
!./kMeans_driver.py C > purities-C.txt
```

In [132]:
```
!./kMeans_driver.py D > purities-D.txt
```

In [ ]:
```
from matplotlib import pyplot as plot
import numpy as np
import re
%matplotlib inline

k = 4
plt.figure(figsize=(15, 15))

## function loads data from any of the 4 initializations
```

```
def loadData(filename):
    purities = []
    f = open(filename, 'r')
    for line in f:
        line  = line.strip()
        data = re.split(",",line)
        iterations.append(int(data[0]))
        purities.append(int(data[0]))
        i = 0
        print data
        for i in range(len(data)):
            purities[i].append(float(data[i]))
            if i:
                purities.setdefault(i,[])
                purities[i].append(float(data[i]))
    return purities

## load purities for initialization A
purities = []
purities = loadData("purities-A.csv")
iterations = [i+1 for i in range(len(purities[1]))]

## plot purities for initialization A
plot.subplot(2,2,1)
plot.axis([0.25, max(iterations)+0.25,0.45, 1.01])
plot.plot(iterations,purities[1],'b',lw=2)
plot.plot(iterations,purities[2],'r',lw=2)
plot.plot(iterations,purities[3],'g',lw=2)
plot.plot(iterations,purities[4],'black',lw=2)
plot.ylabel('Purity',fontsize=15)
plot.title("A",fontsize=20)
plot.grid(True)

## load purities for initialization A
purities = {}
purities = loadData("purities-B.txt")
iterations = [i+1 for i in range(len(purities[1]))]

## plot purities for initialization B
plot.subplot(2,2,2)
plot.axis([0.25, max(iterations)+0.25,0.45, 1.01])
plot.plot(iterations,purities[1],'b',lw=2)
plot.plot(iterations,purities[2],'r',lw=2)
plot.title("B",fontsize=20)
plot.grid(True)

## load purities for initialization C
purities = {}
purities = loadData("purities-C.txt")
iterations = [i+1 for i in range(len(purities[1]))]
```

```
## plot purities for initialization C
plot.subplot(2,2,3)
plot.axis([0.25, max(iterations)+0.25,0.45, 1.01])
plot.plot(iterations,purities[1],'b',lw=2)
plot.plot(iterations,purities[2],'r',lw=2)
plot.plot(iterations,purities[3],'g',lw=2)
plot.plot(iterations,purities[4],'black',lw=2)
plot.xlabel('Iteration',fontsize=15)
plot.ylabel('Purity',fontsize=15)
plot.title("C",fontsize=20)
plot.grid(True)

## load purities for initialization D
purities = {}
purities = loadData("purities-D.txt")
iterations = [i+1 for i in range(len(purities[1]))]

## plot purities for initialization D
plot.subplot(2,2,4)
plot.axis([0.25, max(iterations)+0.25,0.45, 1.01])
plot.plot(iterations,purities[1],'b',lw=2)
plot.plot(iterations,purities[2],'r',lw=2)
plot.plot(iterations,purities[3],'g',lw=2)
plot.plot(iterations,purities[4],'black',lw=2)
plot.xlabel('Iteration',fontsize=15)
plot.title("D",fontsize=20)
plot.grid(True)
```

This experiment shows us how the outcomes of a classification with K-means clustering can depend on the initialization. After initialization B, K-means clustering finds 2 clusters, with purity level 85% and 65%, which may not be a desirable result. In initialization cases A and C, one cluster reaches a 100% purity, another stays around 90%, another decreases to around 70% level, and another is below 50%. The initialization case with the best results is case D, in which the centroids are determined by the sums across the classes. Here, already after 5 iterations, 3 of the clusteras reach above 95% purity.

In [ ]: