DATSCIW261 ASSIGNMENT 4

MIDS UC Berkeley, Machine Learning at Scale

AUTHOR : Rajesh Thallam
EMAIL : rajesh.thallam@ischool.berkeley.edu
WEEK : 4
DATE : 29-Sep-15

HW4.0

a. What is MRJob? How is it different to Hadoop MapReduce?

- **MRJob is python implementation of mapreduce paradigm available to run jobs on multiple platforms. MRJob allows to create data pipeines and chain them through mapreduce steps. MRJob code can be run on local machine, hadoop cluster or AWS EMR**
- **Hadoop MapReduce is a framework to process large data sets with programs running in a distributed and fault-tolerant way. MRJob is python implementation that allows to access Hadoop distributed file system and run MapReduce job on Hadoop cluster. MRJob is a python wrapper over hadoop streaming API and provides a consistent interface to run the programs irrespective of the environment whether local machine, hadoop cluster or cloud without changing the code.**

b. What are the mapper_final(), combiner_final(), reducer_final() methods? When are they called?

- **mapper_final: to define an action to run after the mapper reaches the end of input**
- **combiner_final: to define an action to run after the combiner reaches the end of input.**
- **reducer_final: to define an action to run after the reducer reaches the end of input.**

An example of reducer_final would be finding the top used word in a corpus. The reducer will keep aggregating the counts for each word and reducer final will yield the top used word.

HW4.1

a. What is serialization in the context of MRJob or Hadoop?

Data serialization is process of converting objects into a byte stream (usually compact than original object) for faster transmission of data over a network (interprocess communication), and for writing to persisitent storage. Data serialization formats an be such as json, avro etc. in Hadoop context

b. When it used in these frameworks?

Serialization is used during interprocess ommunication between the tasks of a mapreduce job such as map, combine, shuffle, reduce. This follows remote procedure protocols that uses serialization to make the data object into a byte stream and the following step (receiver) deserializes the byte stream into the original data object.

c. What is the default serialization mode for input and outputs for MrJob?

The default serialization mode in MRJob for input is RawValueProtocol (raw text value), and for output is JSONProtocol (in JSON format).

Preparation for HW4_*

In [ ]:
```
# stop hadoop
!ssh hduser@rtubuntu /usr/local/hadoop/sbin/stop-yarn.sh
!ssh hduser@rtubuntu /usr/local/hadoop/sbin/stop-dfs.sh
```

In [ ]:
```
# start hadoop
!ssh hduser@rtubuntu /usr/local/hadoop/sbin/start-yarn.sh
!ssh hduser@rtubuntu /usr/local/hadoop/sbin/start-dfs.sh
```

In [ ]:

```
# create necessary directories
!hdfs dfs -mkdir /hw4
```

HW4.2

Recall the Microsoft logfiles data from the async lecture. The logfiles are described are located at: [link1] (https://kdd.ics.uci.edu/databases/msweb/msweb.html) [link2] (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

Note: #denotes comments

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.</span>

Assumptions

This part of homework asks to create pre-processed log file. I have an additional step to create file with page urls which will be used for HW4.4.

Exploratory Analysis

In [474]:

```
# let's see what record types are available in the data (to filter during pre-processing)
!cut -f1 -d"," anonymous-msweb.data | sort | uniq -c
```

```
    294 A
  32711 C
      1 I
      4 N
      2 T
  98654 V
```

In [475]:

```
# let's look at sample data starting C (visitor) and V (page)
!egrep -n '^C|^V' anonymous-msweb.data | head -10
```

```
302:C,"10001",10001
303:V,1000,1
304:V,1001,1
305:V,1002,1
306:C,"10002",10002
307:V,1001,1
308:V,1003,1
309:C,"10003",10003
310:V,1001,1
311:V,1003,1
egrep: write error
```

Pre-Process Log File

In [476]:

```
%%writefile preprocessor_log_file.py
#!/usr/bin/env python
import sys
import re

file_name = sys.argv[1]
valid_records = re.compile('^C|^V|^A')

f_urls = open('page_urls.txt','w')
f_log  = open('transformed_msweb_log.out','w')

# read file treating new line as row separator
for line in open(file_name).read().strip().split('\n'):
    # read only if rows start with C or V
    if valid_records.search(line):
        terms = line.split(",")
        # if row starts with C store the visitor id
        if terms[0] == 'C':
            case = terms[1].strip('"')
        # if row starts with V, print the visitor id with page
        # in the format defined
        if terms[0] == 'V':
            print >>f_log, "{},{},{},{},{}".format(terms[0], terms[1], terms[2], "C", case)
        if terms[0] == 'A':
            print >>f_urls, "{},{}".format(terms[1], terms[4])
```

```
Overwriting preprocessor_log_file.py
```

Preparing to run the job

In [477]:

```
# Use chmod for permissions
!chmod a+x preprocessor_log_file.py
```

Driver Function

In [478]:

```
# HW 4.2: preprocess web log files
def hw4_2():
    #cleanup
    ![ -e page_urls.txt ] && rm -f page_urls.txt
    ![ -e transformed_msweb_log.out ] && rm -f transformed_msweb_log.out

    print "pre-processing microsoft web log file data anonymous-msweb.data"
    !./preprocessor_log_file.py anonymous-msweb.data > transformed_msweb_log.out
    !wc -l transformed_msweb_log.out
    !head transformed_msweb_log.out

hw4_2()
```

```
pre-processing microsoft web log file data anonymous-msweb.data
98654 transformed_msweb_log.out
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
```

HW4.3

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

Implementation Approach

1. Stream the pre-processed log file with page and visits created in the pre-processing step HW4.2
2. Map Reduce Step 1
   - mapper: get page_id with count 1 => page, 1
   - combiner: local aggregate for each page and get counts => page, c
   - reducer: get counts for each page with same key => None, (count, page)
3. Map Reduce Step 2
   - reducer: sort the incoming page counts in descending order and fetch top 5 pages most visited
4. Driver sript
   - capture and format the output

NOTE: Reducer in step 2 has limitation that all pages must fit in memory and this may not be scalable. I could not get jobconf make work in local cluster. Otherwise using seondary sort would remove this limitation.

Map Reduce Job

In [479]:

```
%%writefile MostVisited.py
#!/usr/bin/env python
from mrjob.job import MRJob
from mrjob.step import MRStep
from mrjob.conf import combine_dicts

class MRMostVisited(MRJob):
    def steps(self):
        return [
            MRStep(
                mapper=self.mapper_get_pages,
                combiner=self.combiner_count_pages,
                reducer=self.reducer_count_pages),
            MRStep(reducer=self.reducer_find_most_visited)
        ]

    def mapper_get_pages(self, _, line):
        page = line.split(',')[1].strip()
        yield (page, 1)

    def combiner_count_pages(self, page, counts):
        yield (page, sum(counts))

    def reducer_count_pages(self, page, counts):
        yield None, (sum(counts), page)

    def reducer_find_most_visited(self, _, page_count_pairs):
        # each item of page_count_pairs is (count, word),
        # so yielding one results in key=counts, value=page
        results = sorted(list(page_count_pairs), key = lambda x: x[0], reverse = True)[:5]
        for p in results:
            yield int(p[1]),p[0]

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

```
Overwriting MostVisited.py
```

Driver - Command Line

In [480]:

```
# HW 4.3: most frequently visited pages
def hw4_3():
    print "most frequently visited pages"
    print "page    count"
    !./MostVisited.py -r local transformed_msweb_log.out -q

hw4_3()
#!python MostVisited.py transformed_msweb_log.out
```

```
most frequently visited pages
page    count
1008    10836
1034    9383
1004    8463
1018    5330
1017    5108
```

Driver - Hadoop

```
In [ ]: from MostVisited import MRMostVisited
        mr_job = MRMostVisited(args=['-r', 'local', 'transformed_msweb_log.out', 'q'])
        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)
```

Validation using shell command

```
In [481]: !cut -f2 -d"," transformed_msweb_log.out | sort | uniq -c | sort -k1,1nr | head -5 | awk '{print $2, $1}'
```

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

HW4.4

Find the most frequent visitor of each page using mrjob and the output of 4.2 (i.e., transformed log file). In this output please include the webpage URL, webpageID and Visitor ID.

Implementation Approach

1. In the pre-processing step HW4.2, two files were created
   - pre-processed log file with page and visits
   - urls for each page
2. Map Reduce Step 1
   - mapper: get pair of page_id and visitor_id with count 1 => (page,visit), 1
   - combiner: local aggregate for each (page, visit) and get counts => (page,visit), c
   - reducer: get counts for each (page, visit) => (page,visit), c
3. Map Reduce Step 2
   - mapper: to find most frequent visitor for each page emit (page, visit_count), visit
   - partitioner: key field based partitioner to ensure same page goes to the same reducer
   - secondary sort: to find most frequent visitor for a page, first sort on page and then visitor count in descending order
   - reducer_init: load url file to fetch in reducer stage
   - reducer: emit most frequent users for each page with web URL (page, url, visit count, user counts), (list of users)
4. Driver sript
   - set the partitioner and secondary sort job configurations
   - format the output to show top 5 and bottom 5 pages with top 10 users

NOTE: Since every user has visited every once there was a huge output emitted. To reduce the output (and pretty print), I chose to limit top 10 users for each page.

Map Reduce Job

In [482]:

```
%%writefile MostVisitedForEachPage.py
#!/usr/bin/python
from mrjob.job import MRJob
from mrjob.step import MRStep

class MRMostVisitedForEachPage(MRJob):

    # define MRJob steps
    def steps(self):
        return [
            MRStep(mapper=self.mapper_get_pairs,
                   combiner=self.combiner_count_pairs,
                   reducer=self.reducer_count_pairs),
             MRStep(mapper=self.mapper_find_most_visited,
                   reducer_init=self.reducer_frequent_visitor_init,
                   reducer=self.reducer_frequent_visitor)
        ]

    # mapper: get pair of page_id and visitor_id with
    # count 1 => (page,visit), 1
    def mapper_get_pairs(self, _, line):
        terms = line.strip().split(",")
        key = "{0},{1}".format(terms[1], terms[4])
        yield key, 1

    # combiner: local aggregate for each (page, visit)
    # and get counts => (page,visit), c
    def combiner_count_pairs(self, key, counts):
        yield key, sum(counts)

    # reducer: get counts for each (page, visit) => (page,visit), c
    def reducer_count_pairs(self, key, counts):
        yield key, sum(counts)

    # mapper: to find most frequent visitor for each
    # page emit (page, visit_count), visit
    def mapper_find_most_visited(self, key, value):
        terms = key.strip().split(",")
        new_key = "{0},{1}".format(terms[0], value)
        yield new_key, terms[1]

    # reducer_init: load url file to fetch in reducer stage
    def reducer_frequent_visitor_init(self):
        self.urls = { k:v.strip(' "') for k, v in (line.split(",") for line in open('./page_urls.txt').read().strip().spli
t('\n')) }

    # reducer: emit most frequent users for each page with
    # web URL (page, url, visit count, user counts), (list of users)
    def reducer_frequent_visitor(self, key, values):
        terms = key.strip().split(",")
        page = terms[0]
        visits = int(terms[1])
        visitors = list(values)

        k = '{0:<5} {1:<25} {2:<6} {3:<10}'.format(page, self.urls.get(page, 'NA'), visits, len(visitors))
        v = '{0}'.format(",".join(visitors[:10]))
        yield k, v

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

```
Overwriting MostVisitedForEachPage.py
```

In [483]:

```
# Use chmod for permissions
!chmod a+x MostVisitedForEachPage.py
```

Driver - Command Line

In [484]:

```
# HW 4.4: most frequently visited pages
import time

def hw4_4():
    start_time = time.time()

    # command line runner
    print "finding most frequent visitors for each page"
    !./MostVisitedForEachPage.py \
    -r local transformed_msweb_log.out -q \
    --file page_urls.txt \
    --jobconf 'stream.num.map.output.key.fields=2' \
    --jobconf 'map.output.key.field.separator=,' \
    --jobconf 'mapred.text.key.partitioner.options=-k1,1' \
    --jobconf 'mapred.text.key.comparator.options=-k1,1 -k2,2nr' \
    --jobconf 'mapred.output.key.comparator.class=org.apache.hadoop.mapred.lib.KeyFieldBasedComparator' \
    --partitioner 'org.apache.hadoop.mapred.lib.KeyFieldBasedPartitioner' > most_visited_for_each_page.out

    end_time = time.time()

    # format output
    print "Time taken to find most frequent visitors for each page = {:.2f} second.".format(end_time - start_time)
    print "-" * 100
    print "{0:<5} {1:<25} {2:<6} {3:<10}\t{4}".format("page", "url", "visits", "# visitors", "top 10 visitors")
    print "-" * 100
    !head -5 most_visited_for_each_page.out | sed s/\"//g
    !tail -5 most_visited_for_each_page.out | sed s/\"//g
    print "-" * 100

hw4_4()
#!python MostVisited.py transformed_msweb_log.out
```

```
finding most frequent visitors for each page
Time taken to find most frequent visitors for each page = 12.05 second.
----------------------------------------------------------------------------------------------------
page  url                       visits # visitors       top 10 visitors
----------------------------------------------------------------------------------------------------
1026  /sitebuilder              1      3220             10016,10017,10021,10036,10038,10046,10047,10049,10061,10068
1027  /intdev                   1      507              10017,10031,10038,10068,10166,10219,10252,10261,10304,10335
1028  /oledev                   1      93               10017,10297,10545,10699,10818,11191,11508,11570,11908,12147
1029  /clipgallerylive          1      132              10019,10277,10294,10313,10451,10540,10618,10916,10997,11130
1030  /ntserver                 1      1115             10019,10042,10077,10148,10171,10181,10185,10217,10238,10314
1021  /visualc                  1      380              10012,10065,10156,10197,10208,10348,10456,10459,10598,10638
1022  /truetype                 1      325              10013,10074,10109,10141,10346,10875,11089,11181,11218,11328
1023  /spain                    1      191              10014,10067,10068,10622,10788,10880,10980,11183,11333,11377
1024  /iis                      1      521              10015,10117,10151,10166,10252,10290,10303,10329,10351,10388
1025  /gallery                  1      2123             10016,10050,10054,10068,10069,10074,10076,10085,10119,10134
----------------------------------------------------------------------------------------------------
```

Driver - Hadoop Runner

In [ ]:

```
# create necessary directories
!hdfs dfs -mkdir /hw4/hw4_4
!hdfs dfs -mkdir /tmp
```

In [ ]:

```
# find most frequent visitors for each page using mrjob
# running on a local cluser using Hadoop runner
from MostVisitedForEachPage import MRMostVisitedForEachPage
import os

mr_job = MRMostVisitedForEachPage(args=['-r', 'hadoop',
                                 '--hadoop-home', '/usr/local/hadoop',
                                 '--hadoop-bin', '/usr/local/hadoop',
                                 '-o', 'hdfs:///hw4/hw4_4',
                                 '--owner', 'hduser',
                                 '--file', 'page_urls.txt',
                                 '--hdfs-scratch-dir', 'hdfs:///tmp',
                                 '--jobconf', 'stream.num.map.output.key.fields=2',
                                 '--jobconf', 'map.output.key.field.separator=,',
                                 '--jobconf', 'mapred.text.key.partitioner.options=-k1,1',
                                 '--jobconf', 'mapred.text.key.comparator.options=-k1,1 -k2,2nr',
                                 '--jobconf', 'mapred.output.key.comparator.class=org.apache.hadoop.mapred.lib.KeyFieldBas
edComparator',
                                 '--partitioner', 'org.apache.hadoop.mapred.lib.KeyFieldBasedPartitioner',
                                 'transformed_msweb_log.out', '-v'])

output_file = "most_visited_for_each_page.out"

with mr_job.make_runner() as runner, open(output_file, 'a') as f:
    runner.run()
    count = 0
    # stream_output: get access of the output
    for line in runner.stream_output():
        count += 1
        if count == 5:
            break;
        #print mr_job.parse_output_line(line)
        print >>f, line
```

HW4.5

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...)

Using this data, you will implement a 1000-dimensional K-means algorithm 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
(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.

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.

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</span>

Exploratory Analysis/ Validation

In [485]:

```
# check if all the rows have 1000 words
!awk -F"," '{print NF}' topUsers_Apr-Jul_2014_1000-words.txt | sort | uniq -c
```

```
   1000 1003
```

In [486]:

```
# validate total counts in the raw file
invalid_row_count = 0
for line in open(SOURCE_FILE).read().strip().split('\n'):
    tokens = line.split(',')
    if int(tokens[2]) != sum([int(x) for x in tokens[3:]]):
        invalid_row_count = invalid_row_count + 1

if invalid_row_count == 0:
    print "Valid Source File!"

# validate total counts in the summary file
invalid_row_count = 0
for line in open(SUMMARY_FILE).read().strip().split('\n'):
    tokens = line.split(',')
    if '@' in line:
        continue
    if int(tokens[2]) != sum([int(x) for x in tokens[3:]]):
        invalid_row_count += 1

if invalid_row_count == 0:
    print "Valid Summary File!"
```

```
Valid Source File!
Valid Summary File!
```

K-Means Implementation

Assumptions

- used numpy for handling arrays and large dimensions (though I am not sure how it will do in terms of scalbility)

Functionality of MRJob

- mapper_init: load latest centroids files before running the mapper
- mapper: read input stream and emit key = cluster index and value = tuple(features, class counts)
- combiner: read mapper output and combine features for the same cluster and aggregate class counts
- reducer: emit new centroid with class counts

In [487]:

```
%%writefile Kmeans.py
import numpy as np
from mrjob.job import MRJob
from mrjob.step import MRStep

CENTROIDS="/tmp/centroids"

# find the nearest centroid for data point
def MinDist(data_point, centroid_points):
    # calculate euclidean distance
    euclidean_distance = np.sum((data_point - centroid_points)**2, axis = 1)
    # get the nearest centroid for each instance
    minidx = np.argmin(euclidean_distance)
    return minidx

# check whether centroids converge
def stop_criterion(centroid_points_old, centroid_points_new, T):
    return np.alltrue(abs(np.array(centroid_points_new) - np.array(centroid_points_old)) <= T)

class MRKmeans(MRJob):

    centroid_points=np.array([])

    # define mrjob steps
    def steps(self):
        return [
            MRStep(
                mapper_init = self.mapper_init,
                mapper=self.mapper,
                combiner = self.combiner,
                reducer=self.reducer
            )
        ]

    # load centroids from file
    def mapper_init(self):
        self.centroid_points = np.loadtxt(CENTROIDS, delimiter=',')

    # load data and output the nearest centroid index and data point
    # returns key = nearest centroid, values = tuple(features, class:1)
    def mapper(self, _, line):
        terms = line.strip().split(',')
        userid = terms[0]
        code = int(terms[1])
        total = int(terms[2])
        features = np.array([float(x) / total  for x in terms[3:]])

        # key    = centroid
        # values = tuple(features, code:1)
        yield int(MinDist(features, self.centroid_points)), (list(features), {code:1})

    # combine sum of data points locally
    def combiner(self, idx, inputdata):
        combine_features = None
        combine_codes = {}

        for features, code in inputdata:
            features = np.array(features)

            # local aggregate of features
            if combine_features is None:
                combine_features = np.zeros(features.size)
            combine_features += features

            # count number of codes
            for k, v in code.iteritems():
                combine_codes[k] = combine_codes.get(k, 0) + v

        yield idx, (list(combine_features), combine_codes)

    # aggregate sum for each cluster and then calculate the new centroids
    def reducer(self, idx, inputdata):
        combine_features = None
        combine_codes = {}

        for features, code in inputdata:
            features = np.array(features)

            # local aggregate of features
            if combine_features is None:
                combine_features = np.zeros(features.size)
            combine_features += features

            # count number of codes
            for k, v in code.iteritems():
                combine_codes[k] = combine_codes.get(k, 0) + v

        # new centroids
        centroids = combine_features / sum(combine_codes.values())

        yield idx, (list(centroids), combine_codes)

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

```
Overwriting Kmeans.py
```

Driver

Driver script handles initialization and parameterization by accepting as input parameters

1. cluster size, k
2. centroid distribution type, [uniform, perturbed, trained]
   - uniform: uniform random distribution (random centroid)
   - perturbed: normalized noise + aggregated summary (random centroid)
   - trained: normalized class level aggregated value (known centroid)

Following are the steps after initialization

1. Initialize centroid points based on the distribution method
2. Call K-Means mapreduce job to compute new centroids
3. Compare new and old centroid points for convergence
4. If convergence is found, report statistics
5. Else, write new centroids to file and continue from step 2

In [496]:

```
%%writefile hw_4_5.py
import numpy as np
import sys
from Kmeans import MRKmeans, stop_criterion

# initialize variables
SOURCE    = "topUsers_Apr-Jul_2014_1000-words.txt"
SUMMARY   = "topUsers_Apr-Jul_2014_1000-words_summaries.txt"
CENTROIDS = "/tmp/centroids"
```

```
THRESHOLD = 0.001

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

# define mrjob runner
mr_job = MRKmeans(args=[SOURCE])

# validate driver inputs - K and distribution type
if len(sys.argv) != 3:
    print "Invalid number of arguments. Pass k (cluster size) and centroid distribution type (uniform, perturbed, normal)"
    sys.exit(1)

k = sys.argv[1]
try:
    k = int(k)
except:
    raise TypeError("Invalid k. k must be an integer")

distr_type = sys.argv[2]
if distr_type not in ['uniform', 'perturbed', 'trained']:
    print "Invalid centroid distribution type. Type should be uniform, perturbed or trained."
    sys.exit(1)

# generate initial centroids based on initialization and parameterization

# (A) uniform random centroid-distributions over the 1000 words
if distr_type == 'uniform':
    # uniform random distribution
    d = np.random.uniform(size=[k, 1000])
    # total
    t = np.sum(d, axis=1)
    # normalize distribution
    centroid_points = np.true_divide(d.T, t).T

# (B) & (C) perturbation-centroids, randomly perturbed from the aggregated (user-wide) distribution
# perturbation is assumed as noise
elif distr_type == 'perturbed':
    # read ALL_CODES line in the aggregated summary file
    aggregated = open(SUMMARY, 'r').readlines()[1].strip().split(',')

    # normalize
    total = int(aggregated[2])
    summaries = [float(wc) / total for wc in aggregated[3:]]

    # normalized perturbation on the aggregated word counts
    perturbation = summaries + ( np.random.sample(size = (k, 1000)) / 1000 )

    # normalize
    t = np.sum(perturbation, axis=1)
    centroid_points = np.true_divide(perturbation.T, t).T

# (D) "trained" centroids, determined by the sums across the classes
elif distr_type == 'trained':
    summaries = []

    # use trained rows in the aggregated file after
    for line in open(SUMMARY).readlines()[2:]:
        # read trained summary counts
        aggregated = line.strip().split(',')

        # normalize
        total = int(aggregated[2])
        summaries.append([float(wc) / total for wc in aggregated[3:]])

    centroid_points = np.array(summaries)

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

# update centroids iteratively
i = 1
while(1):
    # save previous centoids to check convergency
    centroid_points_old = centroid_points

    with mr_job.make_runner() as runner:
        #print "running iteration" + str(i) + ":"
        runner.run()
        centroid_points = []
        clusters = {}

        # stream_output: get access of the output
        for line in runner.stream_output():
            key, value =  mr_job.parse_output_line(line)
            centroid, codes = value
            centroid_points.append(centroid)
            clusters[key] = codes

    if(stop_criterion(centroid_points_old, centroid_points, THRESHOLD)):
        # display statistics
        print "cluster distribution"
        print "-" * 80
        print "iteration # {}".format(i)
        codes = { 0:'Human', 1:'Cyborg', 2:'Robot', 3:'Spammer' }
        max_class = {}
        print "-" * 80
        print "{0:>5} |{1:>12} (%) |{2:>12} (%) |{3:>12} (%) |{4:>12} (%)".format("k", "Human", "Cyborg", "Robot", "Spamme
r")
        print "-" * 80
        for cluster_id, cluster in clusters.iteritems():
            total = sum(cluster.values())
            print "{0:>5} | {1:>5} ({2:6.2f}%) | {3:>5} ({4:6.2f}%) | {5:>5} ({6:6.2f}%) | {7:>5} ({8:6.2f}%)".format(
                cluster_id,
                cluster.get('0', 0),
                float(cluster.get('0', 0))/total*100,
                cluster.get('1', 0),
                float(cluster.get('1', 0))/total*100,
                cluster.get('2', 0),
                float(cluster.get('2', 0))/total*100,
                cluster.get('3', 0),
                float(cluster.get('3', 0))/total*100
            )
            max_class[cluster_id] = max(cluster.values())
        purity = sum(max_class.values())/1000.0*100
        print "-" * 80
        print "purity = {0:0.2f}%".format(purity)
        print "-" * 80
        break

    # write new centroids to file
    with open(CENTROIDS, 'w') as f:
        for centroid in centroid_points:
            f.writelines(','.join(map(str, centroid)) + '\n')
    f.close()
    i += 1
```

```
Overwriting hw_4_5.py
```

(A) K=4 uniform random centroid-distributions over the 1000 words

In [489]:
```
!python hw_4_5.py 4 uniform
```

```
No handlers could be found for logger "mrjob.runner"
cluster distribution
--------------------------------------------------------------------------------
iteration # 6
--------------------------------------------------------------------------------
    k |       Human (%) |      Cyborg (%) |       Robot (%) |     Spammer (%)
--------------------------------------------------------------------------------
    0 |     0 (  0.00%) |     0 (  0.00%) |    11 (100.00%) |     0 (  0.00%)
    1 |     0 (  0.00%) |    51 (100.00%) |     0 (  0.00%) |     0 (  0.00%)
    2 |     1 (  1.25%) |    37 ( 46.25%) |    38 ( 47.50%) |     4 (  5.00%)
    3 |   751 ( 87.53%) |     3 (  0.35%) |     5 (  0.58%) |    99 ( 11.54%)
--------------------------------------------------------------------------------
purity = 85.10%
--------------------------------------------------------------------------------
```

(B) K=2 perturbation-centroids, randomly perturbed from the aggregated (user-wide) distribution

In [490]:
```
!python hw_4_5.py 2 perturbed
```

```
No handlers could be found for logger "mrjob.runner"
cluster distribution
--------------------------------------------------------------------------------
iteration # 4
--------------------------------------------------------------------------------
    k |       Human (%) |      Cyborg (%) |       Robot (%) |     Spammer (%)
--------------------------------------------------------------------------------
    0 |   751 ( 86.62%) |     3 (  0.35%) |    14 (  1.61%) |    99 ( 11.42%)
    1 |     1 (  0.75%) |    88 ( 66.17%) |    40 ( 30.08%) |     4 (  3.01%)
--------------------------------------------------------------------------------
purity = 83.90%
--------------------------------------------------------------------------------
```

(C) K=4 perturbation-centroids, randomly perturbed from the aggregated (user-wide) distribution

In [491]:
```
!python hw_4_5.py 4 perturbed
```

```
No handlers could be found for logger "mrjob.runner"
cluster distribution
--------------------------------------------------------------------------------
iteration # 7
--------------------------------------------------------------------------------
    k |       Human (%) |      Cyborg (%) |       Robot (%) |     Spammer (%)
--------------------------------------------------------------------------------
    0 |     0 (  0.00%) |    51 ( 96.23%) |     2 (  3.77%) |     0 (  0.00%)
    1 |     0 (  0.00%) |     0 (  0.00%) |     8 (100.00%) |     0 (  0.00%)
    2 |     1 (  1.27%) |    37 ( 46.84%) |    37 ( 46.84%) |     4 (  5.06%)
    3 |   751 ( 87.33%) |     3 (  0.35%) |     7 (  0.81%) |    99 ( 11.51%)
--------------------------------------------------------------------------------
purity = 84.70%
--------------------------------------------------------------------------------
```

(D) K=4 "trained" centroids, determined by the sums across the classes

In [492]:
```
!python hw_4_5.py 4 trained
```

```
No handlers could be found for logger "mrjob.runner"
cluster distribution
--------------------------------------------------------------------------------
iteration # 5
--------------------------------------------------------------------------------
    k |       Human (%) |      Cyborg (%) |       Robot (%) |     Spammer (%)
--------------------------------------------------------------------------------
    0 |   749 ( 93.16%) |     3 (  0.37%) |    14 (  1.74%) |    38 (  4.73%)
    1 |     0 (  0.00%) |    51 (100.00%) |     0 (  0.00%) |     0 (  0.00%)
    2 |     1 (  1.22%) |    37 ( 45.12%) |    40 ( 48.78%) |     4 (  4.88%)
    3 |     2 (  3.17%) |     0 (  0.00%) |     0 (  0.00%) |    61 ( 96.83%)
--------------------------------------------------------------------------------
purity = 90.10%
--------------------------------------------------------------------------------
```

Report

- **Part *D* with known centroids has high purity (90.10%) compared to A, B and C**
- **Part *D* converges faster compared to part *C* concluding using known centroids is better than random walk**
- **In parts *A*, *C* and *D* Cyborgs and Robots do not have a clear majority and have overlaps**
- **Part *B* converges faster than the rest as it has low cluster size but has poor purity of all indicating low quality**

**** -- END OF ASSIGNMENT 4 -- ****