DATASCI W261: Machine Learning at Scale

Midterm Exam

Miki Seltzer (miki.seltzer@berkeley.edu)
W261-2, Spring 2016

In [1]:
```
# We will need these so we can reload modules as we modify them
%load_ext autoreload
%autoreload 2
```

In [2]:
```
# Just in case we need a cluster
# Create job flow so that we don't need to keep spinning up clu
!python -m mrjob.tools.emr.create_job_flow
```

```
using configs in /etc/mrjob.conf
using existing scratch bucket mrjob-ac40f1afcc0b86ce
using s3://mrjob-ac40f1afcc0b86ce/tmp/ as our scratch dir on S3
Creating persistent job flow to run several jobs in...
creating tmp directory /tmp/no_script.cloudera.20160302.234446.
writing master bootstrap script to /tmp/no_script.cloudera.2016
Copying non-input files into s3://mrjob-ac40f1afcc0b86ce/tmp/no_
Waiting 5.0s for S3 eventual consistency
Creating Elastic MapReduce job flow
Job flow created with ID: j-V48FF69EFILM
j-V48FF69EFILM
```

In [3]:
```
clusterId = 'j-V48FF69EFILM'
```

Using the MRJob Class below calculate the KL divergence of the following two objects.

In [4]:
```
%%writefile kltext.txt
1.Data Science is an interdisciplinary field about processes an
2.Machine learning is a subfield of computer science[1] that ev
```

```
Writing kltext.txt
```

MRjob class for calculating pairwise similarity usin 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 [5]:
```
import numpy as np
np.log(3)
```

Out[5]:
```
1.0986122886681098
```

In [99]:
```
%%writefile kldivergence.py
from mrjob.job import MRJob
from mrjob.step import MRStep
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:
            # without smoothing
            #yield key, [index, (count[key]) * 1.0 / (len(lette

            # with smoothing
            yield key, [index, (count[key] + 1) * 1.0 / (len(le


    def reducer1(self, key, values):
        postings = {1:None, 2:None}
        for val in values:
            postings[val[0]] = val[1]
        sim = np.log(postings[1]/postings[2]) * postings[1]
        yield None, sim

    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 [MRStep(mapper=self.mapper1,
                       reducer=self.reducer1),
                MRStep(reducer=self.reducer2)]

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

```
Overwriting kldivergence.py
```

In [98]:
```
from kldivergence import kldivergence

mr_job = kldivergence(args=['kltext.txt', '--no-strict-protocol
print "NO SMOOTHING"

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)
```

```
NO SMOOTHING
(None, 0.08088278445318145)
```

In [100]:
```
from kldivergence import kldivergence

mr_job = kldivergence(args=['kltext.txt', '--no-strict-protocol
print "WITH SMOOTHING"

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)
```

```
WITH SMOOTHING
(None, 0.06726997279170038)
```

MrJob class for Kmeans

In [67]:
```
%%writefile Kmeans.py
from numpy import argmin, array, random
from mrjob.job import MRJob
from mrjob.step import MRStep
from itertools import chain

#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

#Euclidean norm
def norm(x):
    return (x[0]**2 + x[1]**2)**0.5

#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.
               ]

    #load centroids info from file
    def mapper_init(self):
        self.centroid_points = [map(float,s.split('\n')[0].spli
        open('Centroids.txt', 'w').close()

    #load data and output the nearest centroid index and data p
    ##### THIS IS WHERE WE ACCOUNT FOR WEIGHTS #####
    def mapper(self, _, line):
```

In [68]:
```
from numpy import random, array
from Kmeans import MRKmeans, stop_criterion
mr_job = MRKmeans(args=['Kmeandata.csv', '--file', 'Centroids.t

#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 i

# 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
            centroid_points[key] = value
    print "\n"
    i = i + 1
print "Centroids\n"
print centroid_points
```

```
iteration1:
0 [-2.6816121341554244, 0.4387800225117981]
1 [5.203939274722273, 0.18108381085421293]
2 [0.2798236662882328, 5.147133354098043]


iteration2:
0 [-4.499453073691768, 0.1017143951710932]
1 [4.7342756092123475, -0.035081051175915486]
2 [0.10883719601553689, 4.724161916864905]


iteration3:
0 [-4.618233072986696, 0.01209570625589213]
1 [4.7342756092123475, -0.035081051175915486]
2 [0.05163332299537063, 4.637075828035132]


iteration4:
0 [-4.618233072986696, 0.01209570625589213]
1 [4.7342756092123475, -0.035081051175915486]
2 [0.05163332299537063, 4.637075828035132]


iteration5:
```

Go through each data point, find closest centroid, t find weighted distance

In [87]:
```
import csv
from numpy import argmin, array, random

#Euclidean norm
def norm(x):
    return (x[0]**2 + x[1]**2)**0.5

#Calculate find the nearest centroid for data point
def smallestDist(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 distances[min_idx]

data = []

centroids = [[-4.5,0.0],[4.5,0.0],[0.0,4.5]]

num = 0.0
den = 0.0

with open('Kmeandata.csv', 'r') as infile:
    for line in csv.reader(infile):
        point = [float(line[0]), float(line[1])]
        weight = 1/norm(point)
        num += smallestDist(point, centroids) * weight
        den += weight

print num / den
```

```
1.5932559652
```