

**课 程 实 验 报 告**

**课程名称： 大数据分析与处理**

**专业班级： CS1601**

**学 号：U201614531 U201614526**

**姓 名： 刘本嵩 田志伟**

**指导教师： 王蔚**

**报告日期： 2019年11月11日**

**计算机科学与技术学院**

**目 录**

[1 实验方案与总体设计 1](#_Toc5585779)

[1.1概述 1](#_Toc1609626247)

[1.2实验方案 1](#_Toc1206963914)

[1.3实验集群配置 1](#_Toc443790841)

[1.4大数据处理平台配置 2](#_Toc527782585)

[1.5分工说明 2](#_Toc1644736099)

[2 大数据挖掘算法描述 3](#_Toc649877821)

[2.1 BisectingKMeans 算法描述 3](#_Toc1287312596)

[2.1.1 概述 3](#_Toc941499927)

[2.1.2 使用的数据结构 3](#_Toc910668988)

[2.1.3 算法过程分析 4](#_Toc1428837818)

[3 实验过程与结果分析 5](#_Toc42862468)

[3.1 原始数据预处理 5](#_Toc1850727085)

[3.2 分析原始数据以确定算法 5](#_Toc1163005799)

[3.3 BisectingKMeans参数的确定和进行聚类 7](#_Toc1570223670)

[3.4 聚类后的处理 7](#_Toc1355078476)

[3.5 最终推荐结果 8](#_Toc2141344103)

[4 实验结果总结及展望 12](#_Toc1602276018)

[参考文献 14](#_Toc1794488057)

# 1 实验方案与总体设计

## 1.1概述

实验内容为采用BisectingKMeans和DBSCAN这两种聚类算法对数据集 loc-brightkite 的用户登录位置数据进行聚类分析，再根据聚类结果，对行迹较为匹配的好友进行推荐。

## 1.2实验方案

1. 对 loc-brightkite 的totalCheckin数据进行简单的预处理
2. 使用选定的聚类算法进行聚类。由于DBSCAN算法相比于KMeans家族，不易于并行化，可扩展性差，效果的提升也有限。因此主要介绍BisectingKMeans。
3. 聚类算法负责为用户的每一个checkin的位置打上tag，然后每一个用户都可能会有多个tag。我们将tag匹配最长的用户推荐为friend。如果推荐的friends已经达到10个，就不会再返回相似度更低的其他结果了。然后，程序直接生成所有人的好友推荐集合(因为这很快)，如果生产环境有需要，可以将结果输出到任何文件系统或数据库中，以便未来进行快速查询。

## 1.3实验集群配置

Google Cloud Platform, Asia&Pacific, Tokyo Computing Center提供计算资源。

Node Name: login.hpc-lan.recolic.org

Internet address: login.hpc.recolic.org

Intel Xeon @ 2x 2.2GHz, RAM 6GiB

Ubuntu Linux 18.04, Linux kernel 5.0.0-1021-gcp

Hadoop HDFS 3.1.3, Spark 2.4.4 with Hadoop 2.7, Python 3.6.8

Node Name: 1.hpc-lan.recolic.org

Intel Xeon @ 2x 2.2GHz, RAM 6GiB

Ubuntu Linux 18.04, Linux kernel 5.0.0-1021-gcp

Hadoop HDFS 3.1.3, Spark 2.4.4 with Hadoop 2.7, Python 3.6.8

Node Name: 2.hpc-lan.recolic.org

Intel Xeon @ 2x 2.2GHz, RAM 6GiB

Ubuntu Linux 18.04, Linux kernel 5.0.0-1021-gcp

Hadoop HDFS 3.1.3, Spark 2.4.4 with Hadoop 2.7, Python 3.6.8

更详细的环境配置见源码包，或访问Git仓库：<https://git.recolic.org/recolic-hust/hust-bigdata-platformprog>

## 1.4大数据处理平台配置

Hadoop HDFS 3.1.3, Spark 2.4.4 with Hadoop 2.7, Python 3.6.8, GCC 8.3.0 on Linux.

更详细的环境配置见源码包，或访问Git仓库：<https://git.recolic.org/recolic-hust/hust-bigdata-platformprog>

## 1.5分工说明

刘本嵩：平台环境搭建，代码框架实现，代码主体部分实现

田志伟：前期资料搜集，部分代码实现，实验数据测试，撰写实验报告。

# 2 大数据挖掘算法描述

## 2.1 BisectingKMeans 算法描述

### 2.1.1 概述

由于传统的KMeans算法的聚类结果易受到初始聚类中心点选择的影响，因此在传统的KMeans算法的基础上进行算法改进，对初始中心点选取比较严格，各中心点的距离较远，这就避免了初始聚类中心会选到一个类上，一定程度上克服了算法陷入局部最优状态。

算法的主要思想是：首先将所有点作为一个簇，然后将该簇一分为二。之后选择能最大限度降低聚类代价函数（也就是误差平方和）的簇划分为两个簇。以此进行下去，直到簇的数目等于用户给定的数目k为止。以上隐含的一个原则就是：因为聚类的误差平方和能够衡量聚类性能，该值越小表示数据点越接近于他们的质心，聚类效果就越好。所以我们就需要对误差平方和最大的簇进行再一次划分，因为误差平方和越大，表示该簇聚类效果越不好，越有可能是多个簇被当成了一个簇，所以我们首先需要对这个簇进行划分。

### 2.1.2 使用的数据结构

class BisectingKMeans private (

private var k: Int,

private var maxIterations: Int,

private var minDivisibleClusterSize: Double,

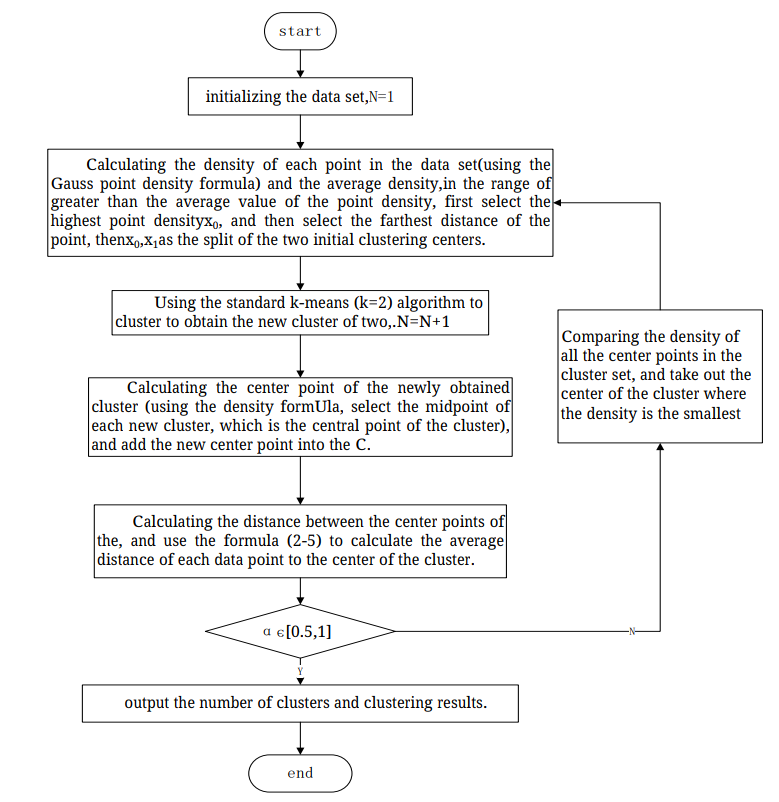
private var seed: Long

)

上面代码中，k表示叶子簇的期望数，默认情况下为4。如果没有可被切分的叶子簇，实际值会更小。maxIterations表示切分簇的k-means算法的最大迭代次数，默认为20。minDivisibleClusterSize的值如果大于等于1，它表示一个可切分簇的最小点数量；如果值小于1，它表示可切分簇的点数量占总数的最小比例，该值默认为1。

### 2.1.3 算法过程分析

BisectingKMeans 算法的计算过程为迭代式过程。迭代过程如下：



# 3 实验过程与结果分析

## 3.1 原始数据预处理

首先对原始的totalCheckins数据进行简单过滤，去除无效的和损坏的数据，去除不必要的列，保留为如下格式，然后放进Amazon S3和Recolic HDFS备用。

DataFile format:

<uid> \t <x> \t <y>

这里提供了3种协议的hosting，你可以用下面的链接访问到预处理后的数据。

<https://aws-logs-936792107402-ap-northeast-1.s3-ap-northeast-1.amazonaws.com/dataset/prepared-totalCheckins.txt>

[s3://aws-logs-936792107402-ap-northeast-1/dataset/prepared-totalCheckins.txt](s3:/aws-logs-936792107402-ap-northeast-1/dataset/prepared-totalCheckins.txt)

[hdfs://login.hpc.recolic.org:54310/prepared-totalCheckins.txt](hdfs:/login.hpc.recolic.org:54310/prepared-totalCheckins.txt)

你可以在任何一台低性能计算机上运行数据预处理的脚本，它很简单，也很快。此脚本源代码如下。

import sys

with open(sys.argv[1]) as f:

lines = f.read()

newLines = []

for line in lines.split('\n'):

if line == '':

continue

try:

ar = line.split()

ar = [ar[0]] + ar[2:-1]

[float(e) for e in ar] # Verify

newLines.append('\t'.join(ar))

except:

print('DEBUG: LINE=' + line)

raise

with open(sys.argv[2], 'w+') as f:

f.write('\n'.join(newLines))

## 3.2 分析原始数据以确定算法

使用GnuPlot对原始数据分布进行绘图，结果的缩略图如图3.1所示。



图3.1 原始数据绘制结果

此图片的超高清版本请访问https://git.recolic.org/recolic-hust/hust-bigdata-platformprog/blob/master/report/bigdata/dataset.png 进行下载。

绘图使用的源代码如下。

# gnuplot.script

set terminal pngcairo enhanced font "Times New Roman,12.0" size 10000,5000

set output '.tmp.png'

set xrange [-180:180]

set yrange [-90:90]

plot '.tmp.txt' with dot

# plot.py

import sys

with open(sys.argv[1]) as f:

lines = f.read()

def printFilter(uid):

return True

newLines = []

for line in lines.split('\n'):

if line == '':

continue

try:

uid, y, x = tuple(line.split())

uid, y, x = int(uid), float(y), float(x)

if not printFilter(uid):

continue

if abs(y) > 90 or abs(x) > 180:

continue

newLines.append(str(x) + ' ' + str(y))

except:

raise

with open(sys.argv[2], 'w+') as f:

f.write('\n'.join(newLines))

# do\_plot.sh

#!/bin/bash

[[ $2 = '' ]] && echo 'Usage: ./do\_plot.sh ~/prepared-totalCheckins.txt output.png' && exit 1

python plot/plot.py "$1" .tmp.txt &&

gnuplot -c plot/gnuplot.script &&

rm .tmp.txt &&

mv .tmp.png "$2"

exit $?

## 3.3 BisectingKMeans参数的确定和进行聚类

首先取不同的分区数，分别进行一次iteration，比较模型cost，结果如下。

|  |  |
| --- | --- |
| Kmeans分区数 | cost |
| 10000 | 75991.63551557351 |
| 30000 | 47039.86338364194 |
| 60000 | 41946.69762908929 |
| 100000 | 40612.03813116302 |

然后决定使用50000作为一个适合的分区数。

然后取不同的迭代次数，观察模型的cost。分别进行5次、10次iterations，结果如下。

|  |  |
| --- | --- |
| 迭代次数(50000个分区) | Cost |
| 1 | 42888.24239110886 |
| 5 | 2362.6930511156947 |
| 10 | 2311.489375889192 |

显然，进行10次iteration基本就足够收敛了。

下面就可以真正进行聚类和打tag了，此部分源代码如下。

model = BisectingKMeans.train(dataInNumpy, partitions, maxIterations=iterations)

taggedActivities = data.map(lambda line: line[0]).zip(model.predict(dataInNumpy)).distinct()

## 3.4 聚类后的处理

聚类算法负责为用户的每一个checkin的位置打上tag，然后每一个用户都可能会有多个tag。我们将tag匹配最长的用户推荐为friend。如果推荐的friends已经达到10个，就不会再返回相似度更低的其他结果了。

大致的处理过程为，先distinct并groupByKey，然后把values中的表拆分为所有可能的组合(flatMapValues)，这里需要限制一下组合数，防止组合爆炸。然后groupByValues，把结果的values部分的values部分剪掉留下key，就得到了tags到users的rdd。此部分源代码如下。

tagsToUsers = taggedActivities.groupByKey().flatMapValues(all\_sublists).groupBy(lambda kv: kv[1]).mapValues(lambda l: [e[0] for e in l])

接下来我们就可以flatMap得到(uid, (friends, tagsLength))这个表，tagsLength就是推荐的weight。这时候直接groupByKey然后mapValues拿出最终结果就可以了。此部分源代码如下。

def recommand(tagsAndUsers):

tags, users = tagsAndUsers

for user in users:

usersWithoutThisOne = list(filter(lambda uid: uid != user, users))

# Results with longer tags are preferred.

yield (user, (usersWithoutThisOne, len(tags)))

def mergeRecommandations(recommandations):

finalResults = []

rs = list(recommandations)

rs.sort(key=lambda ul: ul[1], reverse=True)

for recommand in rs:

users, tagsLength = recommand

finalResults += users

if len(finalResults) > 10:

break

return finalResults

userToFriends = tagsToUsers.flatMap(recommand).groupByKey().mapValues(mergeRecommandations)

## 3.5 最终推荐结果

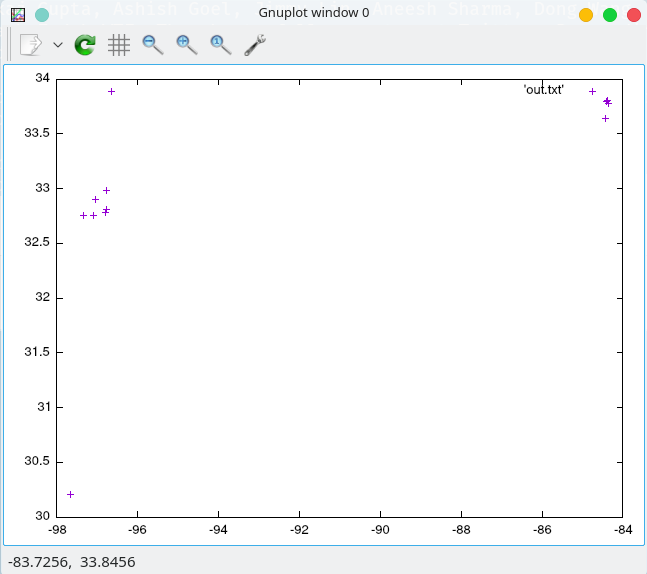
取50000分区，5次iteration的结果，任取其中30个样例，结果如下。

[(125276, [134894, 134724, 106880, 15320, 25920, 10640, 75150, 77790, 103026, 136846, 29328]), (158550, [33054, 14902, 79706, 129476, 61692, 33054, 57404, 79706, 15612, 33054, 118346]), (34394, [56820, 152470, 47116, 130233, 114436, 41078, 5066, 45902, 47794, 78892, 130784, 136038, 147870, 19480, 20847, 104515, 173863]), (162206, [90902, 90902, 154340, 149312, 4274, 22584, 89194, 90902, 116064, 125130, 62279, 149313]), (61528, [41208, 10817, 11363, 58523, 86619, 5812, 9188, 11330, 19162, 34078, 49644, 56494, 64596, 96912]), (8808, [4756, 4756, 4756, 13342, 20308, 33810, 36190, 37742, 46958, 51300, 76494, 79362, 152884, 4075]), (31596, [24009, 88685, 97734, 3968, 20234, 24202, 46960, 62258, 65956, 126768, 24004, 53622, 167310, 989, 2599, 3403, 5163, 7667, 8061, 9287, 11159, 13261, 19269, 28525, 32889, 32909, 33721, 36921, 37967, 45733, 52983, 64639, 69501, 74863, 79901, 90131, 92409, 92463, 94365, 94473, 95509, 97859, 113759, 114323, 118371, 124983, 138301, 143611, 154889, 179913, 193899, 52975]), (178032, [3036, 3378, 41594, 46474, 47524, 65142, 67252, 83342, 90824, 90910, 95868, 126648, 181118, 64537, 154049]), (191548, [14054, 45110, 122270, 126414, 132940, 180224, 2256, 12744, 14066, 16200, 16304, 19782, 20764, 40810, 43890, 51950, 52034, 60846, 60850, 92164, 95890, 101640, 107910, 108422, 123018, 130494, 130502, 130510, 131376, 133972, 139534, 151946, 158346, 167934, 53556, 108748, 146534, 104798, 107606, 122260, 133976, 135090, 45754, 1715, 2131, 4803, 5893, 8391, 12875, 12941, 14801, 15347, 15979, 16195, 16203, 17935, 19221, 20473, 37859, 39555, 39941, 40797, 40811, 43889, 43895, 43909, 45109, 45111, 60839, 60855, 60861, 60863, 68781, 73559, 80875, 91773, 91781, 93905, 94235, 96871, 100129, 101097, 101647, 107157, 109277, 109431, 110821, 111053, 114945, 130503, 137477, 137509, 137527, 138603, 138607, 139429, 149879, 151953, 159005, 168157, 169053, 171289, 73433, 101979, 194609]), (117356, [132250, 111976, 1124, 56424, 58018, 115870, 132250, 153630, 171881, 192601, 6936, 87504, 111976, 145192, 150070, 165616, 10147, 54737, 56619, 102445, 107799, 117113, 132353, 139511, 140199, 142729, 143059, 145393, 146767, 148319, 165745, 166021, 166459, 179047, 184755, 97393]), (179730, [8242, 8242, 11124, 11140, 36422, 47762, 61024, 63864, 68168, 74728, 76546, 89078, 113568, 127090, 132892, 152098, 165172, 65672, 69596, 16180, 41674, 375, 2463, 2811, 3609, 4075, 4135, 6535, 6763, 8201, 8579, 9673, 9867, 10717, 11059, 11205, 11229, 11275, 11345, 12145, 12957, 15183, 16835, 17597, 22013, 22017, 22071, 23333, 24329, 28321, 28693, 29673, 32229, 33731, 39075, 41549, 43519, 44179, 44377, 47765, 48263, 50349, 58527, 58531, 58533, 61015, 63865, 65271, 65671, 66795, 67141, 69229, 69231, 69267, 69599, 73423, 74881, 76435, 76547, 77767, 83427, 84717, 88261, 90165, 90287, 90459, 90523, 90713, 90727, 90875, 92495, 92529, 97737, 113505, 113611, 114111, 115227, 118133, 120195, 123955, 125017, 126433, 154245, 156151, 156321, 167529, 175033, 178013, 189111, 11333, 56087]), (118524, [144298, 144298, 110668, 144298, 135430, 144298, 143080, 58416, 54922, 100530, 144298, 153030, 168378]), (173790, [167794, 87902, 104124, 167794, 18986, 87902, 187838, 87250, 102746, 103368, 97098]), (6898, [133754, 29157, 56252, 56354, 61846, 123622, 45943, 56252, 56354, 59510, 61846, 73618, 80622, 88782, 89010, 101034, 123622, 173556, 6807, 102793]), (18882, [4332, 10302, 992, 3730, 4332, 4850, 9024, 26380, 27846, 29214, 29354, 32974, 34648, 35296, 37490, 41510, 43828, 48338, 53720, 71132, 75220, 75524, 106316, 112882, 116278, 121172, 125328, 144516, 145366, 150780, 196366, 72094, 22975, 39407, 46147, 48869, 60871, 121113, 123823, 126909, 154389, 160227]), (111642, [123496, 157624, 19343, 14512, 74320, 123496, 157624, 4535, 19343, 64453, 75793, 110765, 115867, 143535]), (36318, [126790, 68070, 126790, 17901, 20123, 46801, 55777, 68037, 86775, 86841, 97053, 102657, 103703, 106721, 121831, 136157, 169739, 171685, 179353, 184781]), (107222, [104332, 104332, 48334, 70886, 86186, 104332, 189784, 48334, 70886, 71952, 98008, 104332, 144288]), (83840, [80324, 114496, 80324, 113414, 114496, 125496, 195264, 140476, 72472, 72714, 30635, 32045, 38421, 41563, 46487, 46743, 46935, 57113, 62195, 73073, 76077, 81425, 107305, 122797, 140065, 156847, 161079, 179965, 185137, 195263, 26291]), (97736, [6153, 35811, 59577, 10811, 40057, 78909, 95083, 77796, 269, 3667, 6591, 10811, 13903, 15189, 29231, 35825, 40057, 44339, 44399, 45885, 47469, 50345, 69311, 78909, 90907, 95083, 103749, 124151, 126173]), (166446, [88016, 166580, 182850, 11659, 58367, 58431, 87525, 88003, 90643, 107683, 16398, 17968, 46870, 87522, 88016, 90638, 91444, 91450, 110868, 132550, 138380, 139340, 140624, 147652, 148160, 148572, 149040, 155652, 165026, 166448, 166560, 166580, 175766, 182850, 184538, 186732, 194510, 88006, 189238, 88002, 107682, 112466, 166450, 187456, 10079, 10137, 11659, 17439, 18849, 39195, 39591, 45639, 45933, 53737, 56607, 58367, 58431, 87521, 87525, 88003, 88011, 88015, 90643, 90651, 91445, 91455, 91457, 98485, 99539, 100017, 100775, 102447, 104089, 107207, 107683, 117495, 126131, 126133, 131961, 132069, 132095, 132207, 139689, 141337, 145411, 148211, 148217, 148771, 149247, 150243, 150245, 150461, 164535, 164849, 165021, 165027, 166079, 166445, 166451, 166453, 166891, 168379, 171219, 174233, 175307, 183783, 186785, 187455, 187457, 190771, 191203, 195171, 166579]), (16848, [84601]), (38814, [984, 1082, 1886, 2786, 4570, 4636, 5892, 6726, 7170, 8402, 8422, 8446, 8458, 8462, 8684, 10668, 10676, 10684, 10690, 10692, 10708, 11872, 14186, 14460, 14930, 15008, 15350, 16154, 16822, 17176, 18720, 19074, 20200, 20242, 20342, 20482, 20562, 21958, 23738, 23764, 23766, 23924, 24586, 25144, 25158, 25160, 25758, 27144, 27174, 27368, 28088, 28094, 28484, 29496, 29522, 30646, 30836, 33848, 35048, 35962, 36110, 36596, 36666, 37040, 37254, 37388, 37656, 39298, 39738, 39926, 40060, 40298, 40428, 40814, 41212, 41382, 41444, 43368, 43490, 45174, 45270, 46886, 48278, 48334, 48978, 49282, 49376, 49588, 50002, 50232, 50238, 50252, 50564, 50912, 50948, 53516, 53754, 55308, 55342, 57104, 60296, 60422, 62708, 63104, 63968, 64390, 70160, 70886, 71952, 74172, 75558, 75794, 76260, 80350, 82436, 82824, 86186, 89020, 93870, 94656, 94706, 96052, 96876, 98008, 98174, 99664, 100128, 101346, 102312, 102828, 102884, 102974, 104332, 104814, 106106, 107222, 108458, 109226, 109956, 110090, 110484, 112658, 114256, 114694, 115326, 115668, 116650, 116722, 117162, 117622, 118828, 120090, 120204, 120626, 120726, 121800, 122228, 124802, 127168, 127886, 128630, 129372, 130880, 131010, 131078, 131776, 132310, 132880, 132986, 133606, 136222, 141512, 142910, 142950, 144288, 144658, 146994, 151260, 157104, 157526, 158212, 159024, 159350, 160490, 166748, 168064, 173148, 174190, 178576, 179822, 180452, 187506, 188772, 189784, 189974, 193376, 193504, 53116, 53254, 55876, 115424, 8433, 8459, 9613, 9725, 10097, 10661, 10691, 12589, 16301, 20581, 20823, 23379, 23671, 28873, 29539, 32815, 34735, 37187, 39791, 39935, 40301, 43361, 50115, 50229, 55313, 56147, 59345, 60295, 63921, 69085, 71799, 73317, 76651, 84171, 84173, 84587, 85497, 93411, 95321, 96311, 97693, 104361, 104601, 108827, 109231, 110183, 112625, 113499, 114991, 115357, 116203, 122195, 122301, 122561, 127303, 127887, 129075, 129439, 130091, 130485, 131945, 136465, 143459, 144279, 145577, 147721, 148109, 154763, 156493, 159051, 168737, 169083, 176899, 177863, 178359, 179181, 183205, 186351, 188077, 191935, 195891, 73669, 77387, 80345, 104159, 114695, 50729, 188883]), (185530, [1394, 17872, 57072, 57072, 143245, 1394, 6966, 17872, 46128, 49962, 57072, 115190, 189986, 51831, 61261, 143245]), (125836, [30084, 33812, 72126, 120588, 168916, 124022, 96090, 10384, 41772, 41774, 124022, 159830]), (113298, [10920, 46034, 87622, 139720, 148208, 164252, 146953, 192439, 164634, 46034, 54996, 129660, 147098, 98307]), (174912, [105206, 74112, 76814, 16314, 20366, 39590, 74112, 76814, 96180, 98022, 105798, 134976, 138716, 148774, 163942, 165722, 166090, 174012, 182950, 185990]), (103618, [17866, 100748, 104622, 107368, 107380, 153774, 101441, 175225, 184717, 41982, 41996, 132256]), (187932, [3446, 26060, 29648, 71110, 129738, 133210, 158808, 462, 124796, 143024, 117936, 2001, 4529, 8021, 9833, 29649, 31891, 33723, 52877, 68081, 78813, 83077, 126083, 127679, 157899, 176349]), (79520, [103497, 141363, 5439, 59781, 49944, 141362, 193990, 109658, 17300, 47790, 47798, 47800, 80384, 103498, 104170, 104336, 109656, 141372, 141376, 151058, 158302, 160692, 161102, 165412, 188004, 79526, 141364, 172338, 178742, 185556])]

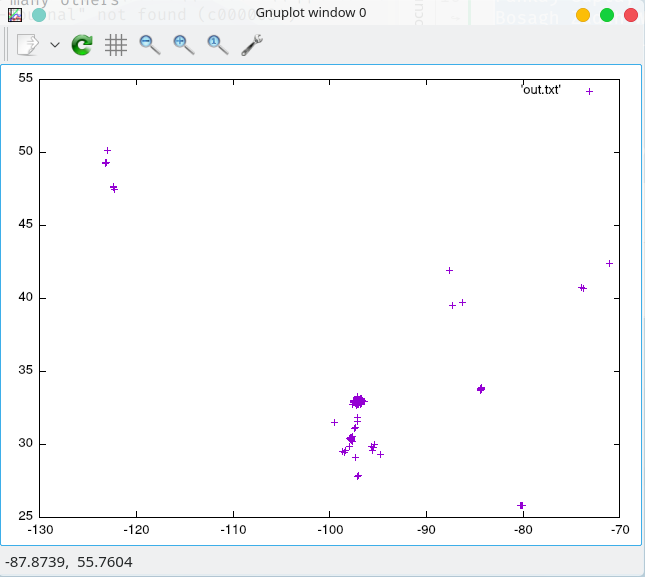
我们以125276用户为例，展示其推荐的朋友的相关性。其结果为，

(125276, [134894, 134724, 106880, 15320, 25920, 10640, 75150, 77790, 103026, 136846, 29328])

我将125276用户的轨迹画在图上，如下图所示。这可能意味着此用户在西边工作，在东边生活。



被推荐的朋友的轨迹如下图所示，请注意看坐标系的转换。显然，被推荐的朋友都来过他的工作/生活区域。它们基本是match的。



# 4 实验结果总结及展望

本次实验是对大数据这门课所学知识的总体检验，难度较大，也让我们从中学习到了很多，巩固了课上所学的知识。

首先，在Amazon AWS搞了一个账号，开了3台高性能的机器。结果提交前几天账号被封了(可能是机器资源用的太多，还一直不充值)，只好在Google Cloud Platform重新开3台机器，重新搭建环境。

集群搭建使用最新版的Hadoop和Spark，在官网的指导下基本没有什么困难，除了OOM导致集群爆炸，和重启脏的HDFS时造成的namenode损坏，加上scala和sparkSdk和库版本和sparkRuntime和javaRuntime的版本问题，还有愚蠢的spark不自带blas库(也不会用openblas，只能重新带netlib编译spark)的问题，也包括ubuntu的.bashrc有-z的坑的问题，同时也见到spark中java依赖管理的弱点(这也是最终抛弃scala的根本原因)，以及其他的一大堆无关紧要的小问题，浪费了不少时间。但其实相比于其他主流的大数据计算平台，Hadoop与Spark的可维护性总体表现是非常优异的。

其实在写这个报告的时候，partition=100000的效果比50000要好的多，只不过金钱不足，没在deadline前跑出这个更好的结果。Anyway，调参也不是这个实验的评分关键。我们能观察到数据，Loss在partition=70000时,就能直接在50000的最终收敛值的基础上再减半。

这个实验我们测试了DBSCAN，PowerIteration，和传统KMeans等算法，包括与高斯混合的结合使用。由于spark的mllib的加成，这部分并没有花费太多精力。在实际效果来看，传统KMeans家族的这些小弟，表现还是比想象中要好的。本来以为会有很多的类似美国加州那样的狭长地形，会给KMeans变种算法造成困难，因此花了不少努力在DBSCAN上，但线性划分的结果最终是很好的，我们最终也就提交了这种算法作为最后结果。因为它简单，所以运算极快。我们的垃圾主频的6核服务器CPU，20分钟就能完成一次50000分区的iteration。(这还是在Spark mllib自带的Blas很垃圾的情况下，专业的超级计算集群，例如Cori，一定要上Intel MKL的。这愚蠢的spark 2.1风格的mllib API居然在用java字节码跑科学计算)

有趣的是，Spark居然带Amazon S3插件，这使得s3://链接能和hdfs://链接一样，很方便的把数据集投送到测试中。这项功能在Amazon EMR上运行Spark任务时非常有用，只可惜，EMR的logging还是有点问题。

Simple is beautiful.

# 参考文献

[1] Seif, G. (2018, February 6). The 5 Clustering Algorithms Data Scientists Need to Know. Retrieved November 11, 2019, from https://towardsdatascience.com/the-5-clustering-algorithms-data-scientists-need-to-know-a36d136ef68.

[2] K. Alsabti, S. Ranka, and V. Singh, “An Efficient k-means Clustering Algorithm”, Proc. First Workshop HighPerformance Data Mining, Mar. 1998.

[3] Pankaj Gupta, Ashish Goel, Jimmy Lin, Aneesh Sharma, Dong Wang, and Reza Bosagh Zadeh WTF: The who-to-follow system at Twitter, Proceedings of the 22nd international conference on World Wide Web

[4] Bangoria Bhoomi, M. "Enhanced K-Means Clustering Algorithm to Reduce Time Complexity for Numeric Values." International Journal of Computer Science and Information Technologies 5.1 (2014): 876-879.

[5] Patil, Ruchika, and Amreen Khan. "Bisecting K-Means for Clustering Web Log data." International Journal of Computer Applications 116.19 (2015).

[6] Apache Hadoop Document. (n.d.). Retrieved November 11, 2019, from https://hadoop.apache.org/docs/stable/index.html.

[7] Spark 2.4.4 Documentation. (n.d.). Retrieved November 11, 2019, from https://spark.apache.org/docs/2.4.4/.