期末报告
云计算

毛羽翎，林雁纯，刘皓青

2022 年 6 月 22 日

目录

1 **实验要求** 2

2 **研究背景及意义** 2

3 **小组分工** 3

4 **实验原理** 3
- 4.1 Hadoop . . . 3
- 4.2 K-Means 算法 . . . 3
- 4.3 MapReduce . . . 4
- 4.4 MapReduce 实现 k-Means 思路 . . . 5

5 **实验过程** 6
- 5.1 环境准备 . . . 6
- 5.2 数据准备 . . . 6
- 5.3 K-Means 类 . . . 9
- 5.4 MapReduce 实现 . . . 12
- 5.5 将代码部署到华为云上 . . . 16
  - 5.5.1 配置环境 . . . 16
  - 5.5.2 运行代码 . . . 16

6 **实验结果** 18

7 **实验总结** 20

8 **个人总结** 20

1 实验要求

利用分布式编程框架 mapreduce，设计并编写一个分布式的程序，用 mapreduce 的思想解决一个实际问题，数据量尽可能大（可以做的方向包括但不限于：文字处理、数据仓库、机器学习等）。在华为云上部署集群并运行程序。

1.1 文字处理，如 Google 的应用：

Distributed sort、document clustering、Web access log statistics、inverted index …

1.2 数据仓库，如 Facebook 的应用：

Reporting: e.g. Daily/Weekly aggregations of impression/click counts

Ad hoc Analysis: e.g. how many group admins broken down by state/country

Collecting training data: e.g. User engagement as a function of user attributes

1.3 机器学习，合适的算法有：

Naïve Bayes、k Nearest Neighbor、kMeans / EM、Random Bagging、Gaussian Mixture…

1.4 近年来 (2017 年之后) 已有的论文中的分布式算法

1.5 近年来 (2017 年之后) 已有的论文中的没有分布式实现过的算法

2 研究背景及意义

在数据挖掘中，聚类是很重要的一个概念，是分析数据并从中发现有用信息的一种有效手段。基于“物以类聚”的朴素思想，使得在同一簇中的对象之间具有较高的相似度，而在不同簇中的对象差别很大，通过聚类，人们能够识别密集和稀疏的区域，发现全局的发布模式以及数据属性之间有趣的相互关系。聚类分析在客户分类、基因识别、文本分类、空间数据处理、卫星照片分析、医疗图象自动检测等领域有着广泛的应用，而其本身的研究也是一个蓬勃发展的领域，数据挖掘、统计学、机器学习、空间数据库技术、生物学和市场学的发展推动着聚类分析研究的进展，使它已成为数据挖掘研究中的一个热点。然而，当数据量过大时，传统的聚类算法将出现以下难题：(1) 由于聚类算法需要将所有数据读至计算机内存中进行处理，这使得面对大数据时其对硬件要求过高。(2) 传统聚类算法收单机 CPU 性能限制，在处理大型数据时速度较慢，效率低下。K-Means 属于聚类分析中一种基本的划分方法，我们选择这个算法，一方面是我们均在其他课程修过相应知识，对其原理和串行计算方法比较熟悉。另一方

面是 k-Means 常采用误差平方和准则函数作为聚类准则, 该算法在处理大数据集时是相对可伸缩且高效率的, 同时具有潜在的数据并行性, 利于我们并行化。

3 小组分工

姓名	学号	分工
毛羽翎、林雁纯	19335156、19335134	将代码部署到 hadoop 集群上
刘皓青	19335137	代码实现

4 实验原理

4.1 Hadoop

Hadoop 是一个开源软件框架，用于在商用硬件的集群上存储数据和运行应用程序。它为任何类型的数据提供了海量存储、巨大的处理能力以及处理几乎无限的并发任务或作业的能力。用户可以在不了解分布式底层细节的情况下，开发分布式程序，充分利用集群的威力进行高速运算和存储。Hadoop 实现了一个分布式文件系统，其中一个组件是 HDFS。HDFS 有高容错性的特点，并且设计用来部署在低廉的硬件上；而且它提供高吞吐量来访问应用程序的数据，适合那些有着超大数据集的应用程序。HDFS 放宽了 POSIX 的要求，可以以流的形式访问文件系统中的数据。Hadoop 的框架最核心的设计就是：HDFS 和 MapReduce。HDFS 为海量的数据提供了存储，而 MapReduce 则为海量的数据提供了计算。
因为我们的实验是用 Mapreduce 实现 k-Means 算法, 然后在 HDFS 上运行, 所以需要部署 Hadoop 平台。

4.2 K-Means 算法

我们选择实现 K-means 算法。
K-means 是我们最常用的基于欧式距离的聚类算法，其认为两个目标的距离越近，相似度越大。主要步骤为：

a. 选取 K 个样本作为初始的聚类中心；

b. 计算每个对象到各个聚类中心的距离，并将其分配给距离最近的聚类中心；

c. 分配过样本后，根据现聚类中的对象重新计算聚类中心；

d. 迭代的中止条件包括没有（或最小数目）对象被重新分配给不同的聚类，没有（或最小数目）聚类中心再发生变化，误差平方和局部最小。

4.3 MapReduce

Hadoop MapReduce 是一个软件框架，基于该框架能够容易地编写应用程序，这些应用程序能够运行在由上千个商用机器组成的大集群上，并以一种可靠的，具有容错能力的方式并行地处理上 TB 级别的海量数据集。

MapReduce 处理数据过程主要分成 Map 和 Reduce 两个阶段。首先执行 Map 阶段，再执行 Reduce 阶段。Map 和 Reduce 的处理逻辑由用户自定义实现，但要符合 MapReduce 框架的约定。MapReduce 处理数据的完整流程如下：

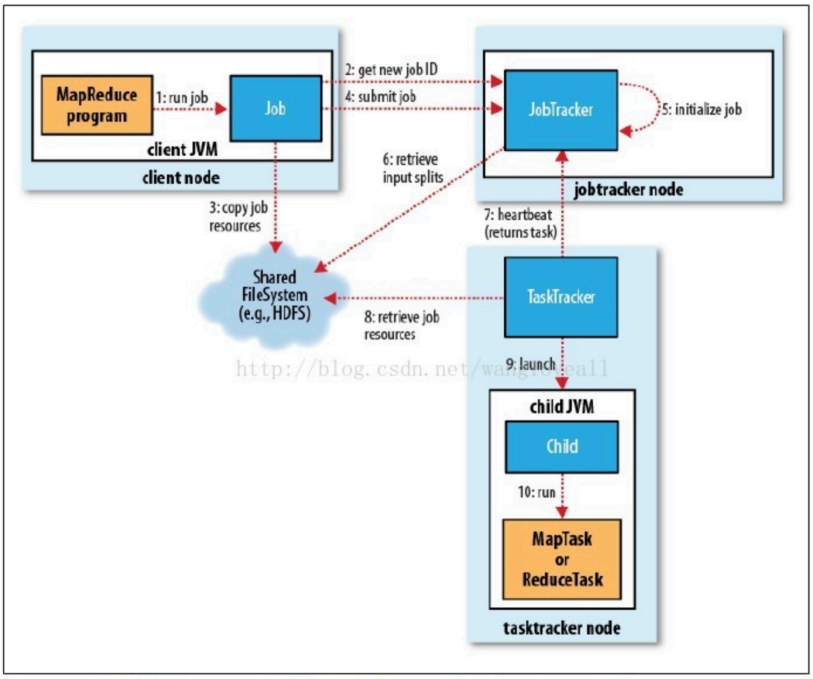

Figure 6-1. How Hadoop runs a MapReduce job using the classic framework

图 1: mapreduce 运行示意图

1. 输入数据：对文本进行分片，将每片内的数据作为单个 Map Worker 的输入。分片完毕后，多个 Map Worker 便可以同时工作。在正式执行 Map 前，需要将输入数据进行分片。所谓分片，就是将输入数据切分为大小相等的数据块，每一块作为单个 Map Worker 的输入被处理，以便于多个 Map Worker 同时工作。

2.Map 阶段：每个 Map Worker 在读入各自的数据后，进行计算处理，最终输出给 Reduce。Map Worker 在输出数据时，需要为每一条输出数据指定一个 Key，这个 Key 值决定了这条数据将会被发送给哪一个 Reduce Worker。Key 值和 Reduce Worker 是多对一的关系，具有相同 Key 的数据会被发送给同一个 Reduce Worker，单个 Reduce Worker 有可能会接收到多个 Key 值的数据。
3. 在进入 Reduce 阶段之前，MapReduce 框架会对数据按照 Key 值排序，使得具有相同 Key 的数据彼此相邻。如果指定了合并操作（Combiner），框架会调用 Combiner，将具有相同 Key 的数据进行聚合。Combiner 的逻辑可以由用户自定义实现。
4.Reduce 阶段：进入 Reduce 阶段，相同 Key 的数据会传送至同一个 Reduce Worker。同一个 Reduce Worker 会接收来自多个 Map Worker 的数据。每个 Reduce Worker 会对 Key 相同的多个数据进行 Reduce 操作。最后，一个 Key 的多条数据经过 Reduce 的作用后，将变成一个值。
5. 输出结果数据。

4.4 MapReduce **实现** k-Means **思路**

实现可行性分析
在进行 K-Means 聚类中，在处理每一个数据点时，只需要知道各个 cluster 的中心信息 (簇 ID, 簇中点个数, 簇中心点对象属性)，不需要知道关于其他数据点的任何信息。数据中所有点对象不互相影响，因此可以进行 Hadoop 并行处理。
MapReduce **并行化** KMeans **伪代码**

```
Begin

    读取inputPath，从中选取前k个点作为初始质心，将质心数
        据写入centerPath；

    While 聚类终止条件不满足

        在Mapper阶段，读取inputPath，对于key所对应的点，
            遍历所有的质心，选择最近的质心，将该质心的编
            号作为键，

        在Combine阶段，刚完成map的机器在本机上都分别完成
            同一个聚类的点的求和，减少reduce操作的通信量
            和计算量。
```

```
        在Reducer阶段，将同一聚类中心的中间数据再进行求
            和，得到新的聚类中心。

    EndWhile

End
```

算法优点

a. 将聚类中心的信息作为全局变量，使聚类归属分配时不需要占用带宽。

b. 利用 combine 本地先进行同一聚类的归并，减少到 reduce 的传输量和计算量。

c. 算法规范，简单。

5 实验过程

5.1 环境准备

需要配置 mrjob 库、sklearn 库。

5.2 数据准备

在这里我们准备的用于聚类的数据产生自 sklearn 下的 make classification 函数。该函数格式为

```
X1, Y1 = make_classification(n_samples=sample_num,
    n_features=feature_dim, n_redundant=0,
                                n_clusters_per_class=1,
                                    n_classes=category_num)
```

这里 n sample 为需要的参数个数, n feature 定义其特征数（维度）, n redundant 定义冗余特征个数。本次实验生成 400 个参数、每个参数 3 个维度，且不需要冗余特征。

```
sample_num = 400
feature_dim = 3
category_num = 3
```

```
X1, Y1 = make_classification(n_samples=sample_num,
    n_features=feature_dim, n_redundant=0,
                                n_clusters_per_class=1,
                                    n_classes=category_num)
```

使用散点图作出聚类结果如下:

```
plt.scatter(X1[:, 0], X1[:, 1], marker='o', c=Y1)
plt.show()
```

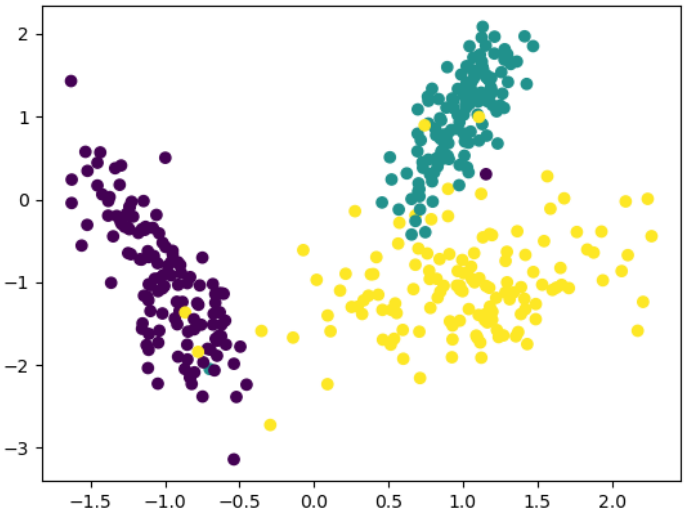

图 2: 随机生成数据的聚类效果示意。

出于对后续实验代码考虑，这里处理数据存储在 txt 文件的格式。

```
#X2 contains items in format(index, category(generated at
    random), feature_dim features)
X2 = np.zeros((sample_num, feature_dim + 2))
for i in range(sample_num):
    X2[i][0] = i + 1
    X2[i][1] = random.randint(1, 3)
    X2[i][2] = X1[i][0]
    X2[i][3] = X1[i][1]
    X2[i][4] = X1[i][2]
np.savetxt('./d.txt', X2, fmt='%.5f', delimiter=',')
```

这里的 d.txt 文件逐行存储一个节点的索引、随机赋予的分类结果和三个维度坐标，在 savetxt 函数中规定这些数字以逗号分隔。

仅以逗号不能分隔 “索引 | 类别 | 坐标” 这三类标签，故还需要做处理，用符号 “|” 分隔。这里输出的数据存储在 data.txt 文件中。

```
#datas.txt contains item in format (index|category(
    generated at random)|feature_dim features)
```

```
#delimiter '|' is used to split
lines = open('d.txt').readlines()
fp = open('data.txt', 'w')
i = 1
for s in lines:
    if i <= 9:
        str = s[:7] + '|'
        str = str + s[8:15] + '|' + s[16:]
    elif i <= 99:
        str = s[:8] + '|'
        str = str + s[9:16] + '|' + s[17:]
    else:
        str = s[:9] + '|'
        str = str + s[10:17] + '|' + s[18:]
    fp.write(str)
    i = i + 1
fp.close()
```

做简单的字符串处理，在原本，的位置以 | 取代。至此，我们完成了数据的准备。

5.3 K-Means 类

创建 KMeans 类，首先设计初始函数:

```
class KMean():
    def __init__(self, data, K):
    """
    K: 集群数;
    centers: 集群中心
    中心初始化方式: 随机选取K个向量作为中心
    """
        self.K = K
        self.dimension = data.shape[1]
        self.centers = np.zeros((self.K, data.shape[1]))
        for i in range(self.K):
            self.centers[i] = data[i]
```

接下来是计算向量到 K 个中心的距离函数：

```
class KMean():
    ...
    def get_distance(self, data):
    distances = np.zeros((data.shape[0], self.centers.
        shape[0]))
    for i in range(data.shape[0]):
        distances[i] = np.sum((self.centers - data[i]) **
            2, axis=1) ** 0.5
    return distances
```

通过向量到 K 个中心的距离比较、argmin 函数得到使得 distance 最小的 index，即为预测所属的集群：

```
class KMean():
    ...
    def get_category(self, data):
        distances = self.get_distance(data)
        category = np.argmin(distances, axis=1)
        avgDistances = np.sum(np.min(distances, axis=1))
            / data.shape[0]
        return category, avgDistances
```

每次都要更新各中心向量，这里通过 update 函数实现：

```
class KMean():
    ...
    def update_centers(self, data, category):
        oneHotClusters = np.zeros((data.shape[0], self.K)
            )
        oneHotClusters[category[:, None] == np.arange(
            self.K)] = 1
        return np.dot(oneHotClusters.T, data) / np.sum(
            oneHotClusters, axis=0).reshape((-1, 1))
```

训练函数 train 调用 update center：

```
class KMean():
    ...
```

```
    def train(self, data):
        category, _ = self.get_category(data)
        newCenters = self.update_centers(data, category)
        diff = np.sum((newCenters - self.centers) ** 2)
            ** 0.5
        self.centers = newCenters
        return diff
```

测试函数调用 get acc 协同实现，得到 accuracy：

```
class KMean():
    ...
    def get_acc(self, predLabels, trueLabels):
        predLabelType = np.unique(predLabels)
        trueLabelType = np.unique(trueLabels)
        labelNum = np.maximum(len(predLabelType), len(
            trueLabelType))
        costMatrix = np.zeros((labelNum, labelNum))
        for i in range(len(predLabelType)):
            chosenPredLabels = (predLabels ==
                predLabelType[i]).astype(float)
            for j in range(len(trueLabelType)):
                chosenTrueLabels = (trueLabels ==
                    trueLabelType[j]).astype(float)
                costMatrix[i, j] = -np.sum(
                    chosenPredLabels * chosenTrueLabels)
        m = Munkres()
        indexes = m.compute(costMatrix)
        mappedPredLabels = np.zeros_like(predLabels,
            dtype=int)
        for index1, index2 in indexes:
            if index1 < len(predLabelType) and index2 <
                len(trueLabelType):
                mappedPredLabels[predLabels ==
                    predLabelType[index1]] =
                    trueLabelType[index2]
        return np.sum((mappedPredLabels == trueLabels).
```

```
            astype(float)) / trueLabels.size

    def test(self, data, labels):
        clusters, avgDistance = self.get_category(data)
        return self.get_acc(clusters, labels),
            avgDistance
```

至此，K-means 算法底层原理部分完成。

5.4 MapReduce 实现

上述过程阐述了 KMeans 类的设计过程，基本过程可以概括为

a. 初始化中心点（这里是随机选取 n 个样本作为初始中心）

b. 计算每个节点到中心的距离，选取 $argmin_n distance$ 作为其归类结果

c. 将所有归为同一类的点坐标求平均值，作为新的中心

为了方便 MapReduce 实现，这里简化 Kmeans 类，将上述每一步分配给 Mapper、Combiner、Reducer 函数。这里开始编写 MRjob 下的 Kmeans 类。MRjob 不限于 Mapper 和 Reducer 函数，还可以覆写 combiner 函数。

```
class KMean(MRJob):
    OUTPUT_PROTOCOL = mrjob.protocol.RawProtocol

    def configure_options(self):
        super(KMean, self).configure_options()

    def read_centroids(self):
        #change the path while running this code
        centroids = np.loadtxt('/Users/myl/Desktop/云计算
            /Centroid.txt', delimiter=',')
        return centroids

    def write_centroids(self, centroids):
        #change the path while running this code
        np.savetxt('/Users/myl/Desktop/云计算/output.txt'
            , centroids, fmt='%.5f', delimiter=',')
```

我们首先定义 read centroids 和 write centroids 两个读、写函数，分别用于读入初始化的中心坐标、将最终的中心写入 txt 文件。

```
class KMean(MRJob):
    ...
    def update_category(self, _, line):
        """
        Mapper function
            calculates distances from items to centroids

            Input: txt, in the format of "ID|category|
                features(dim=3, delimiter=',')"
            Output: new categories of items and their
                indexes and features
        """
        data_index, category, features = line.split('|')
        features = features.strip('\r\n')
        features = np.array(features.split(','), dtype=
            float)
        global centroids
        centroids = self.read_centroids()
        centroids = np.reshape(centroids, (-1, len(
            features)))
        global category_num
        category_num = centroids.shape[0]
        global feature_dim
        feature_dim = centroids.shape[1]
        dist = ((centroids - features)**2).sum(axis=1)
        new_category = str(dist.argmin() + 1)
        feature_list = features.tolist()
        yield new_category, (data_index, feature_list)
```

将该函数用作 Mapper 函数运行，其任务为计算每个节点到中心的距离，接受的输入为格式为”ID|category|features(dim=3, delimiter=’,’)”，即之前处理好的数据格式。distance 为欧氏距离。输出为每个节点的新分类结果、其索引和三维特征坐标。

```
class KMean(MRJob):
```

```
    ...
    def get_new_centroids(self, category, items):
    """
    Combiner function

        calculates the sum of features of the items in
            the same category
    """
    indexes = []
    features = []
    feature_sum = np.zeros(feature_dim)
    for index, feature in items:
        features.append(','.join(str(e) for e in feature)
            )
        feature = np.array(feature, dtype=float)
        indexes.append(index)
        feature_sum += feature
        feature_sum = feature_sum.tolist()
    yield category, (indexes, feature_sum, features)
```

将计算新的中心坐标运用在 combiner 函数，该函数统筹同一类下的点，计算其各维度之和。

```
class KMean(MRJob):
    ...
    def update_centroids(self, category, items):
        """
        Reducer function

            calculates features of new centroids, which
                equal to the averages of features in the
                same category
            updates centroids
            writes into Centroids.txt
        """
        indexes = []
        features = []
```

```
        feature_sum = np.zeros(feature_dim)
        global centroids
        for index, f_sum, fs in items:
            features += fs
            f_sum = np.array(f_sum, dtype=float)
            indexes += index
            feature_sum += f_sum

        curr_centroids = feature_sum / len(indexes)
        centroids[feature_dim * (int(category) - 1) :
           feature_dim * (int(category))] =
           curr_centroids
        if int(category) == category_num:
            #print(centroids)
            centroids = np.reshape(centroids,(1,-1))
            self.write_centroids(centroids)

        for index in indexes:
            idx = indexes.index(index)
            yield None, (index + '|' + category + '|' +
               features[idx])
```

更新中心的函数作为 Reducer。已得各类下点坐标之和后、求算术平均值作为新的中心坐标。输出格式为以｜分隔的”ID|category|features(dim=3, delimiter=’,’)”，以供下次循环 Mapper 读入。

至此，MapReduce 下运行的并行 Kmeans 类设计完成。

```
class Kmean(MRJob):
    ...
    def steps(self):
            return [MRStep(mapper=self.update_category,
               combiner=self.get_new_centroids, reducer=
               self.update_centroids)] * 5
```

在 MRStep 函数中，mapper 函数对应类内的更新聚类函数、combiner 函数对应计算新中心坐标函数、reducer 对应更新中心坐标函数。循环数这里定为常数 5。

5.5 将代码部署到华为云上

5.5.1 配置环境

首先在华为云上搭建 hadoop 集群，实验过程跟上次一样，这里不多加赘述。
因为我们使用 python 完成的程序，因此需要在华为云上安装相关的库，华为云自带的 python 版本为 2.7。
刚开始镜像安装 numpy 失败：

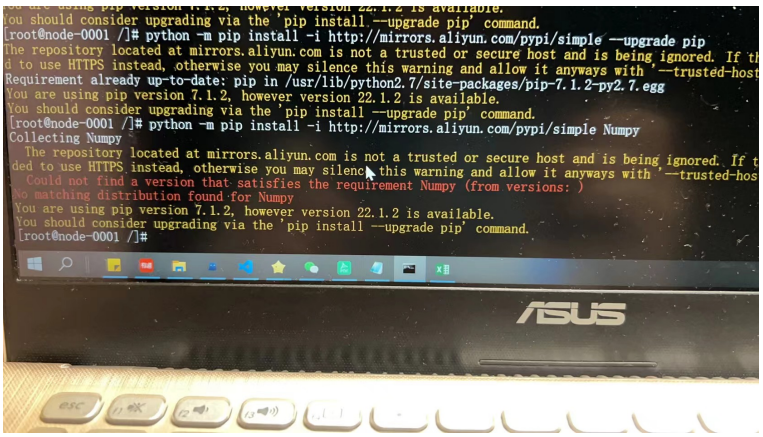

图 3: 安装失败

需要升级 pip，但是升级 pip 也失败了。最后的解决方法是重新在云上安装 python3，然后用 pip3 完成后续安装工作。

5.5.2 运行代码

首先讲代码以及输入数据传到华为云上，这里我们选择直接创建文件然后复制粘贴。
mrjob 的运行模式有以下几种：
1、local 本地测试，直接在本地运行代码，检测代码是否有 bug；
2、inline 内嵌模式，在本地模拟 hadoop 集群上运行，特点是调试方便，启动单一进程模拟任务执行状态及结果，Mrjob 默认以内嵌方式运行（需要着重注意的是 inline 与 hadoop 最终 reducer 的全局排序与局部排序的区别）；
3、hadoop 集群模式，在 hadoop 集群上运行；
4、emr Amazon EMR 模式，参照 aws；
5、dataproc Google Cloud Platform 模式，参照谷歌云平台 Google Cloud Platform。

我们先在本地运行一下自己的程序：
命令如下：
python3 Kmeans_mr.py datas.txt
结果如下：

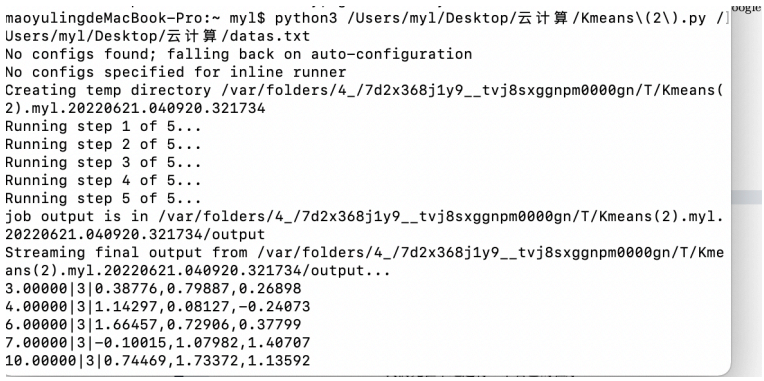

图 4: 本地运行结果

看到代码没有错误，然后再在 hadoop 上运行。
运行时的参数是 python3 + 脚本 + “-r 运行方式” + 数据源 > 输出。数据源可以是本地数据，也可以是 hdfs 上数据，输出可以指定目录。数据源如果是本地，mrjob 会自动上传 hdfs 集群，创建临时文件，待程序运行完成，会自动删除。
这里需要注意提前关闭安全模式，不然会出错：

```
  File "/usr/local/python3/lib/python3.7/site-packages/mrjob/runner.py", line 11
56, in _upload_local_files
    self._copy_files_to_wd_mirror()
  File "/usr/local/python3/lib/python3.7/site-packages/mrjob/runner.py", line 12
57, in _copy_files_to_wd_mirror
    self._copy_file_to_wd_mirror(path, name)
  File "/usr/local/python3/lib/python3.7/site-packages/mrjob/runner.py", line 12
38, in _copy_file_to_wd_mirror
    self.fs.put(path, dest)
  File "/usr/local/python3/lib/python3.7/site-packages/mrjob/fs/composite.py", l
ine 151, in put
    return self._handle('put', path, src, path)
  File "/usr/local/python3/lib/python3.7/site-packages/mrjob/fs/composite.py", l
ine 110, in _handle
    return getattr(fs, name)(*args, **kwargs)
  File "/usr/local/python3/lib/python3.7/site-packages/mrjob/fs/hadoop.py", line
 321, in put
    self.invoke_hadoop(['fs', '-put', src, path])
  File "/usr/local/python3/lib/python3.7/site-packages/mrjob/fs/hadoop.py", line
 183, in invoke_hadoop
    raise CalledProcessError(proc.returncode, args)
subprocess.CalledProcessError: Command '['/home/modules/hadoop-2.8.3/bin/hadoop'
, 'fs', '-put', 'Kmeans.py', 'hdfs:///user/root/tmp/mrjob/Kmeans.root.20220615.0
94121.087972/files/wd/Kmeans.py']' returned non-zero exit status 1.
```

图 5: 运行出错

关闭安全模式：

```
[root@node-0001 code]# hdfs dfsadmin -safemode leave
22/06/15 20:07:16 WARN util.NativeCodeLoader: Unable
ng builtin-java classes where applicable
Safe mode is OFF
```

图 6: 关闭安全模式

运行指令为：

$python3\ Kmeans.py\ -r\ hadoop\ datas.txt > 1.txt$

```
No configs found; falling back on auto-configuration
No configs specified for hadoop runner
reading from STDIN
Creating temp directory /tmp/Kmeans.root.20220615.091312.036333
```

图 7: hadoop 上运行

6 实验结果

centroids 随迭代次数变化如下：

对分类结果作图：

```
[[ 0.78791872  1.29270574 -1.3216966 ]
 [-0.51025741 -1.28915324 -0.17824389]
 [-0.4095978   0.88817229  0.36145061]]
[[ 0.97655474  1.29696923 -0.77497359]
 [-0.75106291 -1.13813063 -0.1986174 ]
 [-0.50878646  0.93594779  0.47619318]]
[[ 1.09943735  1.21646765 -0.40967471]
 [-0.82884739 -1.07354634 -0.20515649]
 [-0.72868378  0.9689411   0.46623848]]
[[ 1.19179473  1.15997545 -0.15070991]
 [-0.84528742 -1.07554121 -0.23215098]
 [-0.89955224  0.96913474  0.35069718]]
[[ 1.18110195  1.11486059 -0.02633449]
 [-0.80660426 -1.10032155 -0.25216008]
 [-1.00487033  0.9772468   0.27987869]]
[[ 1.17461792  1.1042745   0.02708942]
 [-0.7188448  -1.15530561 -0.23412724]
 [-1.08893834  0.9472642   0.20848561]]
[[ 1.15847902  1.09614738  0.04288754]
 [-0.70593909 -1.17625017 -0.21848   ]
 [-1.11046427  0.94093707  0.18082242]]
[[ 1.15847902  1.09614738  0.04288754]
 [-0.67574195 -1.21114085 -0.16240593]
 [-1.12514981  0.92697169  0.13198087]]
[[ 1.15847902  1.09614738  0.04288754]
 [-0.63591612 -1.2316281  -0.10525853]
 [-1.14811883  0.91524512  0.08742611]]
[[ 1.15108557  1.10298     0.04636967]
 [-0.56802491 -1.24534281 -0.09144982]
 [-1.18356512  0.89351433  0.0728872 ]]
[[ 1.13303774  1.09638984  0.06152815]
 [-0.49769327 -1.27048836 -0.07678264]
 [-1.22998494  0.86103741  0.04820434]]
```

图 8: centroids 随迭代次数变化

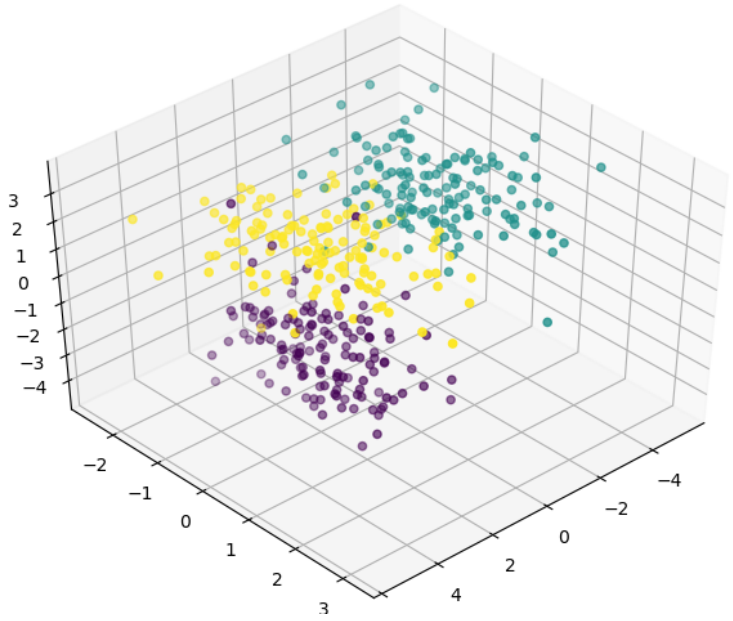

图 9: 分类结果

7 实验总结

a. 在部署的步骤中，华为云自带的版本是 python2.7,python 一些库下载不了，pip 版本过低且无法升级 pip，我们尝试了多种方法，还是失败。最终我们重装了 python3.4，使用 pip3 下载相应库。

b. 代码运行的时候需要创建文件，但是一直报错，我们在本地跑的时候是没有任何问题的。通过查阅大量资料，发现是华为云上安全模式没有关闭，导致无法创建新文件。

c. 华为云的官方配置参考资料太少，导致我们遇到问题时，无法查找权威资料，只能参考博客尝试解决。配置环境这个环节浪费了我们大量的时间，由于我们一开始性能设置过高，导致账号代金券用完，只能更换另一个账号重新配置。

8 个人总结

本次实验目的是完成 Kmeans 这一聚类算法的 MapReduce 实现，这就要求我先去了解算法本身、其次去了解 MapReduce 的编程规则。KMeans 这一算法通

过朴素地计算每个点到中心的欧氏距离、取距离较近的为其分类中心，非常直白、好理解；在了解 MapReduce 编程主体框架分为 Mapper 和 Reducer 函数后，我在 Mapper 中将每个归为同一类的点的 key 设为类别索引，在 Reducer 中对同一类（key 值）的点坐标求和、从而完成节点更新。整个过程较顺利，且算法与 MapReduce 编程规则较契合。