实现基于 Hadoop 的 KMeans 算法

> 李易非 3160105705
>
> 熊苗 3160104714
>
> 李灿晨 3160105187

一、邮件数据预处理

我们获取的邮件原始数据是十分混乱的，其中包含了多种内容模版（具体构成参考我们在 WordCount 试验中作出的论述）。因此，我们希望在本部分中对邮件数据进行预处理，提取邮件文本中表示真正内容的单词，对多余的内容进行去除。相比于 WordCount 中进行的数据预处理，我们在本次的数据预处理中使用了 Python 中用于自然语言处理的相关库 `spacy`，对结果进行了 **word stemming** 和 **stop word** 去除，从而使结果更为准确。此部分的源代码将会在附录中提供。

正文提取

在数据预处理中，我们首先需要提取邮件的正文部分，其中包括了移除邮件 Header 以及移除可能存在的 HTML 标签。我们使用以下方法获取邮件的 Content-Type，并依照邮件的 Content-Type 对其进行相应的处理。

```
def read_mail_content(original_content):
    mail_content = ''
    content_type = get_type_from_content(original_content)
    if content_type is None:
        mail_content = fetch_text_from_plain_text(original_content)
    elif content_type.__contains__('text/plain'):
        mail_content = fetch_text_from_plain_text(original_content)
    elif content_type.__contains__('text/html'):
        mail_content = fetch_text_from_text_html(original_content)
    elif content_type.__contains__('multipart'):
        mail_content = fetch_text_from_multipart(original_content)
    return mail_content
```

单词构成提取

与之前使用正则表达式进行单词识别不同，在本次试验中，我们使用 nlp 对邮件正文进行单词提取，并生成由邮件中的单词组成的特殊数据结构。

```
import spacy
nlp = spacy.load("en_core_web_sm")
nlp_mail_content = nlp(mail_content)
```

我们实现了 `isNoise` 方法，对单词数据结构中的项目进行识别：如果此项目的词性属于我们规定的噪声词性范畴，或者相应单词为 stop word，或者相应单词的结构类似于邮件地址/网页链接，或者单词长度小于2，都会被 `isNoise` 方法判断为 `True` ，我们将会对此类单词进行去除。

```
def isNoise(token, noisy_pos_tags, min_token_length=2):
    is_noise = False
    if token.pos_ in noisy_pos_tags or token.is_stop == True or
token.like_url == True or token.like_email == True or
len(token.string.strip()) <= min_token_length:
        is_noise = True
    return is_noise

## 去除词根，去除stop word，去除噪声词
cleanlist = [utils.cleanup(str(word.lemma_)) for word in nlp_mail_content
if not\
             utils.isNoise(word, noisy_pos_tags) and re.match(r"^[a-zA-
Z]*$", str(word))]
```

依据邮件数据集中提供的 spam/ham 标签文件，我们对垃圾邮件以及非垃圾邮件进行了单独处理，并将相应的处理结果分开存放，以后后期的词向量提取工作。

```
def preprocess_data(nlp, noisy_pos_tags):
    ham_count = 0
    spam_count = 0
    with open(spam_ham_separator_path, 'r') as spam_ham_separator_file:
        while True:
            line = spam_ham_separator_file.readline()
            if not line:
                return
            is_spam, file_name = get_mail_info(line)
            clean_content = get_file_word_list_string(
                file_name, nlp, noisy_pos_tags)
            if is_spam:
                save_as_file(clean_content, True, spam_count)
                spam_count += 1
            else:
                save_as_file(clean_content, False, ham_count)
                ham_count += 1
            print('\r File processed: ' + str(ham_count + spam_count),
end='')
```

二、文档向量提取

在完成了邮件数据处理之后，我们获得了能够代表邮件内容的单词集合。但是，对于 K-Means 算法来说，其输入应该是向量。因此，我们需要通过词向量提取步骤来将之前获得的单词集合映射到一个维度统一的向量，为 K-Means 算法做好准备。此部分的源代码将会在附录中提供。

本质上说，我们只需要使用特定的向量来代表单词集合即可，因此，独热码的解决方案是首先被想到的。但是，对于文章量达 70K 以上的数据集合来说，独热码的解决方案将会导致词向量长度过大，因此，我们需要使用更为复杂的词向量映射方案来减少词向量的维度。

词向量计算

Gemsim 中的 word2vec 作为 Google C 语言版 word2vec 的 Python 封装，能够使用 Continuous Bag-of-Words Model (CBOW) 预测词语出现的概率模型，而模型的参数可以视为对应的文章的词向量被我们使用。我们仅需调用 `gemsim.Word2vec`，即可提取对应文档的词向量。在以下代码中，`size` 参数用来表示最终训练完成的词向量维度；`min_count` 参数用来表示参与到运算中的单词所出现的最小次数，小于此次数的单词不会被计算，这将会显著减少模型的运算量；`iter` 参数用来表示模型中随机梯度下降的循环次数：更高的循环次数通常会带来更高的准确率，但同时也会增加运算时间，有时还会造成过拟合的问题。

```
iteration = 1000
model = Word2Vec(wholeword_list, size=100, min_count=10, iter=iteration)
model.save('./model_iter1000')
vocab = model.wv.vocab
word_vector = {}
for word in vocab:
    word_vector[word] = model[word]
```

在此处，`wholeword_list` 为之前清理过的所有邮件的单词列表。模型训练所消耗的时间是相当长的，在本次试验中，我们在一台 MacBook Pro 上进行训练工作，所消耗的时间大约7小时。在训练完成之后，我们得到了一个单词 - 向量字典，并将其命名为 `vocab`，在之后的工作中，我们只需要在次字典中索引，便可以得到给定单词（如果在数据集中出现）的向量表示。

利用 `word2vec`，通过一晚上的训练，我们得到了邮件数据集所有单词的向量。

文章向量计算

在获得邮件数据集中每个单词的向量表示之后，我们便可以通过进一步的计算来使用向量来表示每一封邮件。将用于表示单词的向量转化使之得以表示邮件有多种方法，本次试验中我们采用了单词向量平均的做法，即找出给定邮件中每一个单词所对应的单词向量，再将这些向量加起来，除以邮件中的单词个数。这种做法虽然损失了邮件中单词的顺序信息，但是依然能够粗略地完成我们的目标，即使用维数相同的向量来表示每一封邮件，从而为 K-Means 算法准备数据。

```
whole_docvec_list = []
for doc in mail_list:
    doc_vec = np.zeros([1, 100])
    for word in doc:
        if word in model:
            doc_vec += model[word]
    doc_vec = doc_vec / len(doc)
    whole_docvec_list.append(doc_vec)
```

在文件输出时，我们在向量之前加入了一位序号标识，从而使得我们能够在聚类完成之后依据序号标识来找到类中的向量所对应的邮件文档。

此部分完成之后，我们将 `whole_docvec_list` 中的结果输出到文档，至此，我们已经准备好 K-Means 算法所需要的文档向量输入。

```
with open("doc_vec", "w") as outfile:
    for idx, docvec in enumerate(whole_docvec_list):
        outfile.write(str(idx) + " ")
        [outfile.write(str(item) + " ") for item in docvec[0]]
        outfile.write("\n")
```

三、 KMeans 算法实现

本部分中，我们将会在 MapReduce 计算框架下实现 KMeans 算法。在此计算框架下，整个计算过程被分为 Map 阶段和 Reduce 阶段，针对 KMeans 算法，我们规定了两阶段中所进行的工作：在 Map 阶段中我们会对每个向量计算与其最相近的聚类中心，在 Reduce 阶段我们会针对每个类中拥有的向量，重新计算给定类的中心。

Map 过程

在 `KmeansMapper` 类中，我们将会实现 Hadoop 的 `Mapper` 接口。根据我们的设定，作为参数的 `key` 是类的序号， `value` 是以字符串表示的一篇邮件的向量。我们通过读取 `configuration` 中的数据来获取每个聚类中心的位置，再通过欧式距离的大小来确定与给定的向量最为接近的聚类中心。`KmeansMapper` 的实现如下：

```
public static class KmeansMapper extends Mapper<Object, Text, IntWritable,
Text> {
    public KmeansMapper(){}
    public void map(Object key, Text value, Context context)
        throws IOException, InterruptedException {
        int DIM = 100;
        StringTokenizer itr = new StringTokenizer(value.toString());
        ArrayList<Float> fields = new ArrayList<Float>(DIM);
        for(int i = 0; i <= DIM; i++) {
            fields.add(Float.parseFloat(itr.nextToken()));
```

```
        }
        Configuration config = context.getConfiguration();
        List<ArrayList<Float>> centers =
DeserializeCenters(config.get("centers"));
        int k = centers.size();
        float minDist = Float.MAX_VALUE;
        int centerIndex = 0;
        for (int i = 0; i < k; ++i) {
            float currentDist = 0;
            for (int j = 1; j <= DIM; ++j) {
                float tmp = Math.abs(centers.get(i).get(j - 1) -
fields.get(j));
                currentDist += Math.pow(tmp, 2);
            }
            if (currentDist < minDist) {
                minDist = currentDist;
                centerIndex = i;
            }
        }
        context.write(new IntWritable(centerIndex), new Text(value));
    }
}
```

Reduce 过程

我们将会在`KmeansReducer`和`ClusterReducer`类中实现 Hadoop 的`Reducer`接口。此处作为参数的`key`应是某个类的序号，而`value`应是此类中所有向量对应的邮件序号以及向量本身的数据，不同向量之间使用换行进行分隔，向量之间的数字使用空格进行分隔。本部分中，我们将会依据向量的内容通过求平均数的方法来更新对应的类的聚类中心，并将计算得出的中心，并将此聚类中心作为 value 写入 context 中。

两个类的区别仅限于其输出结果：在`KMeansReducer`类中，我们会将重新计算的聚类中心作为 value 输出到 context，而在`ClusterReducer`类中，我们会将聚类中心向量以及其邮件数量作为 key 输出，再将包含在其中的向量序号作为 value 输出。鉴于两个类的上述区别，`ClusterReducer`仅用于迭代收敛后的循环中，即最后一个循环中，而在其余时候，都会调用`KmeansReducer`。

`KmeansReducer`的实现如下：

```
public static class KmeansReducer extends Reducer<IntWritable, Text, Text,
Text> {
    public KmeansReducer(){}
    public void reduce(IntWritable key, Iterable<Text> value, Context
context)
        throws IOException, InterruptedException {
        List<ArrayList<Float>> assistList = new ArrayList<ArrayList<Float>>
();
        String tmpResult = "";
        Integer count = 0;
        for (Text val : value) {
```

```
            count += 1;
            String line = val.toString();
            String[] fields = line.split(" ");
            List<Float> tmpList = new ArrayList<Float>();
            for (int i = 0; i < fields.length; ++ i) {
                tmpList.add(Float.parseFloat(fields[i]));
            }
            assistList.add((ArrayList<Float>)tmpList);
        }
        // Get the average vector
        for (int i = 1; i < assistList.get(0).size(); ++i) {
            float sum = 0;
            for (int j = 0; j < assistList.size(); ++j) {
                sum += assistList.get(j).get(i);
            }
            float tmp = sum / assistList.size();
            if (i == 1) {
                tmpResult += tmp;
            } else {
                tmpResult += " " + tmp;
            }
        }
        Text result = new Text(tmpResult);
        // Output the center vectors, note that there would be multiple
files
        context.write(new Text("") , result);
    }
}
```

`ClusterReducer` 的实现如下：

```
public static class ClusterReducer extends Reducer<IntWritable, Text, Text,
Text> {
    public ClusterReducer() {}
    public void reduce(IntWritable key, Iterable<Text> value, Context
context)
        throws IOException, InterruptedException {
        List<ArrayList<Float>> assistList = new ArrayList<ArrayList<Float>>
();
        String tmpResult = "";
        Integer count = 0;
        for (Text val : value) {
            count += 1;
            String line = val.toString();
            String[] fields = line.split(" ");
            List<Float> tmpList = new ArrayList<Float>();
            for (int i = 0; i < fields.length; ++ i) {
                tmpList.add(Float.parseFloat(fields[i]));
```

```
            }
            assistList.add((ArrayList<Float>)tmpList);
        }
        // Get the average vector
        for (int i = 1; i < assistList.get(0).size(); ++i) {
            float sum = 0;
            for (int j = 0; j < assistList.size(); ++j) {
                sum += assistList.get(j).get(i);
            }
            float tmp = sum / assistList.size();
            if (i == 1) {
                tmpResult += tmp;
            } else {
                tmpResult += " " + tmp;
            }
        }
        String cluster = "聚类中心: ";
        cluster += tmpResult;
        cluster += "\n" + "有" + count + "封邮件" + "\n";
        cluster += "对应邮件编号: " + "\n";
        String indexList = "" + assistList.get(0).get(0);
        for (int j = 1; j < assistList.size(); ++j) {
            indexList += " " + assistList.get(j).get(0);
        }
        context.write(new Text(cluster) , new Text(indexList));
    }
}
```

在 `run` 方法中，依据由 `main` 方法提供的用于表示是否是最后一轮迭代的 `runReduce` 参数，我们可以选取两个类中实现的 `reduce` 方法的调用时机。

```
if(runReduce){
    job.setReducerClass(KmeansReducer.class);
} else {
    // Output is required only at the last iteration
    job.setReducerClass(ClusterReducer.class);
}
```

迭代控制

与之前的 WordCount 项目使用一次 MapReduce 即可解决问题不同，在 KMeans 聚类问题中，我们在每一轮迭代中都需要用到一个相对独立的 MapReduce，即将给定向量归并到相应的聚类中心，再重新计算每个聚类中心新的位置。但 KMeans 要求我们不断进行此迭代直到聚类中心不再发生改变，因此，在我们实现的 KMeans 算法中，我们需要进行多轮的 MapReduce 过程，每一次迭代中对聚类中心进行一次更新。

如前文所述，在首次迭代中，我们将随机产生的 k 个聚类中心的向量值保存在 `centerPath` 目录下，是一个单独的文件。而在经过一轮 MapReduce 产生新的聚类中心之后，由于 MapReduce 的特性，输出文件是多个文件，被保存在 `newCenterPath` 目录下。为了控制迭代过程，我们使用 `compareCenters` 方法来对比前后两次迭代过程中聚类中心的变化，此方法可以反馈出聚类中心的变化。此方法将删除 `newCenterPath` 目录下的所有文件，如果聚类中心几乎没有改变，此方法将会保持 `centerPath` 文件不变，否则此方法将会将 `newCenterPath` 目录中的内容整合成单一的文件写入 `centerPath` 中。此方法通过布尔类型输出来表示聚类中心是否发生变化。

```
public static boolean compareCenters(String centerPath,String newPath)
    throws IOException{
    List<ArrayList<Float>> oldCenters = Utils.getCenters(centerPath,
false);
    List<ArrayList<Float>> newCenters = Utils.getCenters(newPath, true);
    int size = oldCenters.size();
    float distance = 0;
    for(int i=0;i<size;i++){
        distance += getDist(oldCenters.get(i), newCenters.get(i));
    }
    if(Math.abs(distance) < 0.001){
        Utils.deletePath(newPath);
        return true;
    } else {
        Configuration conf = new Configuration();
        Path outPath = new Path(centerPath);
        FileSystem fileSystem = outPath.getFileSystem(conf);
        FSDataOutputStream overWrite = fileSystem.create(outPath,true);
        overWrite.writeChars("");
        overWrite.close();
        Path inPath = new Path(newPath);
        FileStatus[] listFiles = fileSystem.listStatus(inPath);
        for (int i = 0; i < listFiles.length; i++) {
            FSDataOutputStream out = fileSystem.create(outPath);
            FSDataInputStream in = fileSystem.open(listFiles[i].getPath());
            // What is this step doing?
            IOUtils.copyBytes(in, out, 4096, true);
        }
        Utils.deletePath(newPath);
    }
    return false;
}
```

根据 `compareCenters` 的输出结果，我们在 `main` 方法中进行迭代控制：首先对 MapReduce 过程进行无限循环，在每次循环的末尾调用 `compareCenters` 来判断迭代中心是否改变同时整合存储迭代前后聚类中心的相应文件，如果迭代中心几乎没有变化，则在调用一次 `ClusterReduce` 实现的 `reduce` 方法将结果输出之后退出无限循环并结束程序。

```
public static void main(String[] args)
```

```
    throws ClassNotFoundException, IOException, InterruptedException {
    String centerPath = "/user/hadoop/centerpath/center";
    String dataPath = "/user/hadoop/datapath";
    String newCenterPath = "/user/hadoop/new_centerpath";
    int count = 0;
    while(true){
        run(centerPath,dataPath,newCenterPath, true);
        System.out.println(" 第 " + ++count + " 次计算 ");
        if(Utils.compareCenters(centerPath, newCenterPath)){
            run(centerPath,dataPath,newCenterPath,false);
            break;
        }
    }
}
```

四、 聚类分析

在上述过程成功运行通过后，我们会得到一个输出结果文件，输出结果格式如下：

```
聚类中心：0.4129776 -2.2857752 0.8921562 -0.9775333 ... (聚类中心的坐标，100维的向量)
有3250封邮件
对应邮件编号：
    24672.0 24673.0 24686.0 24687.0 24688.0 24700.0 24715.0 24749.0 24750.0 24804.0 ... (对应的邮件编号，共3250封)
```

聚类关键词分析

`tf` 全称 `term-frequency`，是一个单词在某一文档中的出现次数，比如在 `I love hadoop and hadoop.` 中 `hadoop` 的 `tf` 就是 2。但一个单词对于一篇文档的重要性不仅仅体现在 `tf` 上。比如上述句子中的 `and` ，会在很多的文档中出现，那么它便不足以标示一个具体的文档，这就引出了 `idf`，全称 `inverse-documents-frequency` 逆文档频率。`idf` 的计算公式为：

$$idf(word) = log(\frac{N}{M+1})$$ 共有N篇文档　　单词出现在M篇文档中

在这一部分中，我们将一个聚类视为一个文档，若有10类，则有10个“文档”。对这10类文档进行 `tf - idf` 方法的计算，以 `tf * idf` 的值作为一个单词在对应聚类中的权重，选出每个聚类前10权重的单词作为聚类关键词。

处理函数如下：

```
def pipeline(PATH):
    ## parse the result_file
    centers_mails_idx = parse_result(PATH, UNIT_LINES)
    clusters_tf_dict = build_tf_from_clusters(centers_mails_idx)
    clusters_idf = build_idf_from_all_clusters(clusters_tf_dict)
    clusters_tfidf = build_tf_idf_of_clusters(clusters_idf,
clusters_tf_dict)
    return clusters_tfidf
```

五、结果分析

输出格式

```
====================
('code', 5914.740595335999) ('write', 3582.569716349638) ('minimal',
3420.8693873163825) ('reproducible', 3410.525813942159) ('posting',
3297.8165316575196) ('read', 2910.0150526886096) ('perl',
2771.2290089304997) ('function', 2560.9260974800986) ('provide',
2543.390197879363) ('contain', 2512.596387798002)
====================
```

结果输出聚类出来k个类的关键词，以键值对 `{word: tf*idf}` 的形式体现。

以下展示我们以 `k = 10` 得到的10个类的关键词：

1. 程序语言类

```
==================== 程序代码类
('branch', 6562.105618584798) ('struct', 6512.07823142962) ('lib',
6321.502286706701) ('char', 5578.448509146439) ('const', 5521.567950273658)
('heimdal', 4292.163047421962) ('rev', 3676.412014243835) ('null',
3630.7049317729934) ('utc', 3619.199544382044) ('modified',
3541.2889454807605)
==================== 程序交流类
('code', 5914.740595335999) ('write', 3582.569716349638) ('minimal',
3420.8693873163825) ('reproducible', 3410.525813942159) ('posting',
3297.8165316575196) ('read', 2910.0150526886096) ('perl',
2771.2290089304997) ('function', 2560.9260974800986) ('provide',
2543.390197879363) ('contain', 2512.596387798002)
```

我们聚出两类的结果相对比较相似，都涉及到高级程序语言的一些关键词，如 `struct char const null` 等。通过查看相关类别的原文，我们发现这两类邮件内容不过也有一些区别。第一类更多是直接贴源码，第二类是对程序的讨论。

其中`UTC`为世界统一时间，`perl`是给予Web开发的高级编程语言。

2. 减肥药类

```
====================减肥药类
('anatrim', 15156.81363365921) ('christians', 2403.1297174345686)
('weight', 2393.7288729674324) ('ennis', 1674.7261708173771) ('hgh',
1601.676199316023) ('try', 1565.3520124691815) ('great',
1518.4918666931974) ('pill', 1460.2272204851704) ('product',
1430.573307475399) ('world', 1420.7549912175737)
```

```
anatrim：对抗饥饿和燃烧脂肪的药物
christians基督教
Ennis 爱尔兰小镇
HGH human growth hormone
```

关键词中除了一些燃烧脂肪的药物`anatrim`和生长素`HGH`之类，还有一些体重`weight product great pill`等词汇，所以我们判断这是售卖减肥药产品的广告。

3. 壮阳药类

```
====================壮阳药类
('canadianpharmacy', 10026.798194464445) ('viagra', 6638.689806462816)
('anatrim', 4242.800162444599) ('erection', 3951.632076372248) ('pharmacy',
3678.9661423626653) ('cialis', 3469.37717969205) ('nma', 3206.000321568728)
('generic', 3112.9713512299472) ('erectile', 2889.064677599211)
('herbalking', 2856.752294570528)
====================
('pill', 14872.988487301202) ('itemyour', 4752.670155417898) ('item',
2389.2604135725283) ('viagra', 1280.1290131575727) ('tabs',
733.5455957279627) ('cialis', 675.8990187638979) ('levitra',
662.6487046153077) ('jelly', 526.7918572255584) ('soma', 334.0969410298936)
('soft', 326.90530267531733)
```

词汇解释：

```
canadianpharmacy 加拿大药房
erection：勃起
generic ：  无注册商标
erectile：无勃起功能的
herbalking：国际采购商
====================
viagra：万艾可：治疗勃起功能障碍的伟哥
tab：迷幻药
cialis:西力士：治疗男性勃起功能障碍
Levitra ：for the treatment of male erection problems
jelly：a powerful hypnotic drug
soma：a muscle relaxer that blocks pain sensations between the nerves and
the brain.
```

这也是两个类相似性非常高。我们可以通过一些药物产品的解释可以看到这是加拿大壮阳药产品的广告邮件。

4. 公司激活邮件

```
====================公司商业邮件
('desjardins', 26017.279422317548) ('votre', 14688.4819032458) ('accsd',
14370.671120124081) ('vous', 12814.90507419227) ('pas', 6742.935772487148)
('mouvement', 5763.4687034884355) ('groupe', 5749.724342510323)
('courrier', 5322.4111764195695) ('rpondez', 5100.308744503664) ('cliquez',
5064.9011104301135)
```

```
其中`tf-idf`值最高的`desjardins`是一家加拿大银行，`Accsd`是它们的线上服务`Online
Solution`。
```

以下这些词汇都是法语，通过查阅得到词汇解释：

```
votre 您的
courrier 快递
pas 不
rpondez 答复
cliquez 点击click
remercie 谢谢
constatez 找到
communiquez 接触
```

通过对这些敬语和文件的分析，我们猜测这可能是该银行在法国的分部回复用户的商业邮件。

5. 艺术设计类

```
====================软件设计类（如Adobe系列
('retail', 5151.456215307112) ('adobe', 3583.1564868085056) ('macromedia',
2599.3019270997947) ('acrobat', 1913.086218345449) ('photoshop',
1780.6951068584995) ('professional', 1506.238288253267) ('microsoft',
1159.4538926286339) ('autodesk', 1102.104017090313) ('dummies',
1072.739768654409) ('illustrator', 1043.1865067427177)
macromedia 多媒体
acrobat Adobe的PDF创建程序
Autodesk 软件设计公司
dummy 傻瓜
```

通过`Adobe Acrobat Microsoft Macromedia Autodesk`等著名互联网应用公司以及`professional illustrator retail`等词汇我们知道这是一个互联网应用的零售推销邮件。

6. 金融投资类

```
====================金融投资类
('subscribe', 1766.8321632473005) ('ntelos', 1611.0473503465344) ('oneok',
1530.2494342982648) ('netflix', 1473.6627124949457) ('relations',
1419.3343436730663) ('alert', 1418.7466992557456) ('email',
1369.4319744153054) ('satcon', 1247.3158252816697) ('nasd',
1194.3410218913286) ('investor', 1078.5370129512748)
====================
('broker', 1669.0984107883482) ('chvc', 1631.9700432081777) ('bullish',
1305.714292920671) ('asvp', 1218.344499712614) ('brocker',
1052.2722309808682) ('aggresive', 860.3969972298316) ('target',
855.9786628413086) ('bvyh', 743.5603155445543) ('chvccurrent',
724.2470605953451) ('sym', 636.9995528361904)
```

专业词汇解释：

```
nTelos 无线电信公司
ONEOK 能源公司
netflix 奈飞
SatCon 能源公司
nasd 纳达斯克
====================
broker 股票经纪人
chvc 晨晖创投 or 胜伟西通信
bullish 看涨的
asvp ：Advanced Strategic Value Propositions/(management consulting)
brocker：买卖人/代理人
CHVCcurrent 某家资本一只股票
BVYH： Brunton Vineyards Holdings Inc (BVYH) 资本公司
SYM：摩托车公司
```

这两类的相似性也非常高。高频出现的有股票经纪人、纳达斯克以及一系列的上市公司，所以我们猜测这是一类讨论商业股票的公司。

7. 新闻广播

```
====================新闻广播
('cbs', 5373.128851710046) ('font', 4355.001065491922) ('cnn',
3561.390213716999) ('president', 3460.883872535807) ('weather',
2874.4813577820933) ('country', 2803.411023227757) ('payment',
2754.882588969988) ('energy', 2584.266830632147) ('dept', 2536.22553366884)
('nbsp', 2335.947992649668)
```

词汇：

```
CBS 加拿大广播公司
font 字体
CNN 美国有线电视新闻网
weather 天气
Nbsq Noaaport Broadcast System Processor
```

这一类高频词汇都是跟新闻广播公司以及天气预报等，所以很自然我们可以得出这是新闻广播类。

六、Kmeans 运行时间与 hadoop 节点的关系

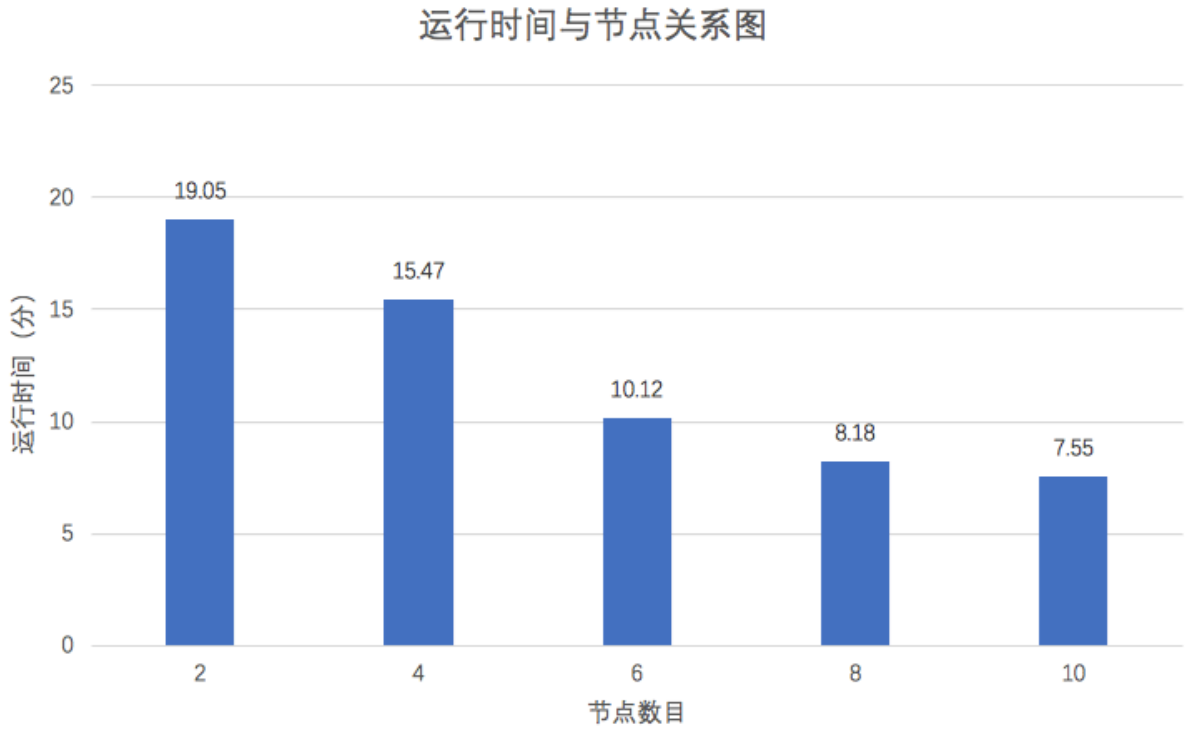
新增服务器过程中遇到的问题

首先我们尝试利用 `hadoop 2.7.6` 进行新增服务器搭建及测时，但出现了很多很多奇怪的问题，最后改来改去回到了 `retrying to connect to server` 的胶着状态。

最终我们决定采用 `hadoop 3.1.1` ， `hadoop 3.1.1` 与 `hadoop 2.7.6` 差别不大，部分 `.xml` 配置不同，但也大同小异，唯一需要特别注意的是， `slaves` 文件夹改名为 `workers` , 在 workers 文件下指定 slave 服务器名。新增服务器流程记录如下：

1. 先在原 hadoop 机群中进行 `stop-all.sh` 命令；
2. 从原有镜像中恢复一个 slave 服务器，并将 master 服务器的公钥填入 `authorized_keys` 中；
3. master 服务器修改 `/etc/hosts` 文件，加入新武器的 `{IP 名称}` 的映射，并通过 `scp` 命令分发给各个 slave 服务器；
4. 进入新服务器，修改 `hdfs-site.xml` 文件，将 property `dfs.data.dir` 属性进行修改，并清除之前镜像服务器的 `data` ；
5. 进入 master 服务器， `start-all.sh` ，并通过 `hdfs dfsadmin -report` 命令来检查所有节点是否成功开启；

结果



从运行时间和节点的关系可以看出，随着节点的增加运行时间逐渐变快，但是变快的程度并非线性，反而随着节点的增多，变快程度越来越慢，最后趋于平稳。

七、遇到的问题及解决方案

在本次工程中，我们的 KMeans 算法以及相应的分析方法的构建并不是一帆风顺的，经历了许多问题。为此，我们尝试了众多解决方案，并最终获得了令人满意的聚类结果以及可以接受的聚类速度。

MapReduce 过程中的性能问题

在 KMeans 算法的初次尝试中，我们实现的算法遇到了较为严重的性能问题：更新单词聚类中心1次迭代需要消耗超过20分钟的时间，这个性能对于拥有数台服务器的集群来说显然是不可接受的。

我们第一个怀疑的是 Java 代码中的 `split` 函数，认为此函数的频繁调用可能会导致性能受到局限：此函数在每次将以字符串表示的向量组转化为浮点数阵时都将会被调用，我们所检索到的一些文件也支持我们此想法。但是，经过我们的实验，发现性能的瓶颈并不在 `split` 函数上：使用该函数分隔百维向量的单词运算被限制在毫秒以内。类似的，我们也怀疑了我们新建字符串的方式：是否应该使用 `StringBuilder` 而不是直接实例化空的字符串，而相关的尝试说明这也不是问题的关键。

我们想到了在上次试验中部分小组碰到的问题：如果将邮件数据集中的每一封邮件分开放置，则会使集群运算的性能大幅降低。这个例子从一个侧面说明大量的 I/O 操作会严重拖慢 hadoop 集群运算的速度，基于这个模糊的想法，我们重新检查了整个代码，发现了端倪。在 `map()` 进程中，我们需要读入 k 个 centers 的坐标，我们原本的处理方法如下:

```
List<ArrayList<Float>> centers =
Utils.getCenters(context.getConfiguration().get("centerpath"));
```

`Utils` 类中封装了一系列实验中需要用到的函数， `Utils.getCenters(PATH)` 函数定义如下:

```
ArrayList <ArrayList<Float>> result = new ArrayList <ArrayList<Float>> ();
Path path = new Path(centersPath);
Configuration conf = new Configuration();
FileSystem fileSystem = path.getFileSystem(conf);
FSDataInputStream fsis = fileSystem.open(path);
LineReader lineReader = new LineReader(fsis, conf);
Text line = new Text();
while(lineReader.readLine(line) > 0) {
      ArrayList<Float> tempList = textToArray(line);
      result.add(tempList);
}

lineReader.close();
return result;
```

大致思路就是，通过传入的路径 `PATH`，通过 `FSDataInputStream` 和 `LineReader` 打开文件，读出 centers 信息，而我们的速度瓶颈也就恰恰在此，每一篇文档对应的向量都要进行一次 `map()` 过程，70000多篇文档， centers 文件将会被打开 70000 多次，大量的 I/O 操作限制了 hadoop 集群运算的速度。

改正的思路也比较自然，通过 `Configuration.set()` 方法，将 centers 信息存入 `Configuration()` 即可，这样 70000 多个 `map()` 获得 centers 信息的时候直接从内存中读取，不需要进行 I/O 操作。

经过上述的修改思路，一次迭代的时间从 **30+** min 降到了 **1** min。