实验（一）

1. 预处理文本数据集，并且得到每个文本的 VSM 表示
2. 实现 KNN 分类器，测试其在 20Newsgroups 上的效果

一、 安装 Git

安装 git bash
创建 GitHub 仓库

git add 添加文件到仓库
git commit 提交文件到仓库
git remote add origin git@github.com:qtli/201814816-liqintong.git 将本地库与远程库关联
git pull origin master 取回远程主机的更新
git push origin master 将本机 master 分支的修改提交远程主机

二、 理论

为了得到每个文本的 VSM 表示，需要以下步骤：

1. 对每个文档预处理，包括：Tokenization，Normalization，Stopwords filtering，词频过滤，最终建立词典
2. 统计文档总数，每个单词出现过的文档数，文档中每个单词的出现数，计算 TF，IDF
3. 对 tf_idf 公式 log 规范化

$$tf(t,d) = c(t,d)$$

$$IDF(t) = \log\left(\frac{N}{df(t)}\right)$$

4. 计算一个文档中所有词的 tf_idf，计算所有文档对所有词的 tf_idf

KNN 分类器步骤如下:

1. 划分训练集和测试集，得到所有文本对所有词的 tf_idf
2. 对于每一个测试文档，向量化，cosine similarity 计算与训练文本的距离

$$cosine(d_i, d_j) = \frac{V_{d_i}^T V_{d_j}}{|V_{d_i}|_2 \times |V_{d_j}|_2}$$

3. 在训练文本中选出与测试文本最相近的 K 个文档，依此计算每个类的权重
4. 比较每个类的权重，将测试文本预测为权重最大的类

三、 代码

1. 构建字典

read_allfiles(path)：读取全部文件
clean_lines(s)：句子内容清理，去掉数字标点和非字母字符
clean_words(tokens)：去掉英文停用词
word_count(cleans)：计算每个单词出现的频数
build_dict(wc,5)：构建词典，最小词频设定为 5

```
path = '/home/liqintong/code/VSM/20news-18828'
# 得到全部的文件列表（全路径）
allfiles = read_allfiles(path)
tokens = []  # 全部单词
for file in allfiles:  # 对每个文件处理
    file = open(file, 'rb')
    for line in file.readlines():
        raw = str(line)  # 原始文本 byte -> string
        sens = raw.split('\\n')  # 分割成句子，得到文本对换行符的句子列表
        for s in sens:  # 对每个文件的每个句子处理
            s1 = clean_lines(s)  # 清理句子内容
            for t in s1.split():  # 分词
                tokens.append(t)
cleans = clean_words(tokens)  # 去停用词(得到词袋)
wc = word_count(cleans)  # 计算词数
wd = build_dict(wc, 5)  # 参数是词数和最小词频，构建最终词典
print('字典构建完毕，字典大小为：%d' % len(wd))
```

最终运行结果如下：

```
(myknn) [liqintong@localhost VSM]$ python KNN1.py
字典构建完毕，字典大小为：36630
```

2. 计算每个词的 IDF，并写入文件保存

```
# 将IDF值写入文件保存
start = time.clock()
idf = computeIDF('/home/liqintong/code/VSM/20news-18828')
end = time.clock()
print('computeIDF runtime: %s', str(end - start))

fw = open('/home/liqintong/code/VSM/IDFPerWord', 'w')
for word, idf in idf.items():
    fw.write('%s %.6f\n' % (word, idf))
fw.close()
```

computeIDF(path)：计算每个词的 IDF，

```
for i in range(len(cateList)):
    sampleDir = fileDir + '/' + cateList[i]    # 遍历第一层目录
    sampleList = listdir(sampleDir)   # 遍历第二层目录
    for j in range(len(sampleList)):
        docNum += 1
        sample = sampleDir + '/' + sampleList[j]   # 文件的全路径
        tokens = get_tokens(sample)
        cleans = clean_words(tokens)
        for word in cleans:
            if word in wordDocMap.keys():
                wordDocMap[word].add(sampleList[j])  # set结构保存单词word出现过的文档
            else:
                wordDocMap.setdefault(word, set())
                wordDocMap[word].add(sampleList[j])
    print("IDF finish %d", i)
```

wordDocMap 存储每个词与存在该词的文档，格式为 {word,(doc1,doc2,…)}
IDFPerWordMap 存储每个词及对应 IDF 值，格式为{word1:idf1,word2:idf2…}

a. 遍历每个文档得到单词与其对应的文档，并计算出现文档总数，通过 log()正则化，计算得到每个词的 IDF 值，返回 IDFPerWordMap 字典；
b. 将 IDFPerWordMap 字典中的 key 和 value 写入文件保存，方便之后使用。

3. 计算每个文档中每个词的 TF_IDF 值

```
start = time.clock()
computeTFIDF(0, 0.75)
end = time.clock()
print('computeTFIDF runtime: %s', str(end - start))
```

computeTDIDF(0,0.75)计算 TF_IDF 值，参数为开始索引和训练集比例
针对训练集和测试分别创建两个文件存储结果，

```
for j in range(len(sampleList)):
    TFPerDocMap = {}   # <word, 文档doc下该word出现次数>
    sumPerDoc = 0  # 记录文档doc下的单词总数
    sample = sampleDir + '/' + sampleList[j]
    tokens = get_tokens(sample)
    cleans = clean_words(tokens)
    for word in cleans:
        sumPerDoc += 1
        TFPerDocMap[word] = TFPerDocMap.get(word, 0) + 1  # 得到类别i，文档j的字典，key为文档的每个词，value为词在文档出现的次数

    if (j >= testBeginIndex) and (j < testEndIndex):
        tsWriter = tsTestWriter
    else:
        tsWriter = tsTrainWriter

    tsWriter.write('%s %s ' % (cateList[i], sampleList[j]))   # 写入类别cate，文档doc
```

TFPerDocMap 存储每个单词及其在每个文档中出现的次数
根据 index 值判断该文档属于训练集还是测试集，写入对应的结果文件

```
    # TFPerDocMap.pop('b')
    tfidf.append(cateList[i])
    tfidf.append(sampleList[j])
    for word, count in TFPerDocMap.items():
        TF = float(count)/float(sumPerDoc)    # 计算TF值
        tfidfValue = float(TF) * float(IDFPerWord[word])  # 类别i下的文档j的所有有效词的tfidf值tfidfValue = TF * float(IDFPerWord[word])
        tsWriter.write('%s %f ' % (word, tfidfValue))   # 写入类别cate下的文档doc下的所有单词及它的TF-IDF值
        tfidf.append((word, tfidfValue))

    tsWriter.write('\n')
print("TF_IDF finish %d", i)
```

tfidf 存储文档类别，文档，单词及其 idf 集合，即类别 i 下文档 j 的所有有效词的 tfidf 值集合，格式为[cate, doc, (word1,tfidf1),(word2,tfidf2),…]
最终训练集文档和测试集文档结果均被写入对应的文件中。

4. KNN 算法为测试文档分类
K 经过多次实验，最终设置为 20，结果较好

```
start = time.clock()
knnProcess(20)
end = time.clock()
print('KNN runtime: %s', str(end - start))
```

分别读取训练集文档和测试集文档的有效词 tfidf 集合，生成 trainDocWordMap 和 testDocWordMap 字典，key=cate_doc，value={(word1,tfidf1),(word2,tfidf2)…}

```
trainDocWordMap = {}  # 字典<key, value>   key = cate_doc, value = {{word1. tfidf1}, {word2, tfidf2},...}
for line in open(trainFiles).readlines():   # 打开训练文件
    lineSplitBlock = line.strip('\n').split(' ')
    trainWordMap = {}
    m = len(lineSplitBlock)-1
    for i in range(2, m, 2):   # 在每个文档向量中提取（word, tfidf）存入字典，组成字典的value值   range(start, stop, step)
        trainWordMap[lineSplitBlock[i]] = lineSplitBlock[i+1]

    temp_key = lineSplitBlock[0] + '_' + lineSplitBlock[1]  # 在每个文档向量中提取类别cate，文档doc，变成cate_word的形式
    trainDocWordMap[temp_key] = trainWordMap


testDocWordMap = {}
for line in open(testFiles).readlines():
    lineSplitBlock = line.strip('\n').split(' ')
    testWordMap = {}
    m = len(lineSplitBlock)-1
    for i in range(2, m, 2):    # 步长为2
        testWordMap[lineSplitBlock[i]] = lineSplitBlock[i+1]

    temp_key = lineSplitBlock[0] + '_' + lineSplitBlock[1]
    testDocWordMap[temp_key] = testWordMap   # <cate_doc, <word, tfidf>>
```

调用 KNNComputeCate()对测试文档分类，返回 sortedCateSimMap[0][0]，得到与测试文档向量距离和最小的类；

KNNComputeCate()调用 computeSim(testDic,trainDic)返回余弦相似度，余弦值越大，、越相似。

```
# 遍历每个测试样例计算与所有训练样本的距离，为测试样例分类
count = 0
rightCount = 0
KNNResultWriter = open(KNNResultFile, 'w')
for item in testDocWordMap.items():
    classifyResult = KNNComputeCate(item[1], trainDocWordMap, K)
```

比较测试文档真实所属类与 KNN 算法分类结果，计算准确率

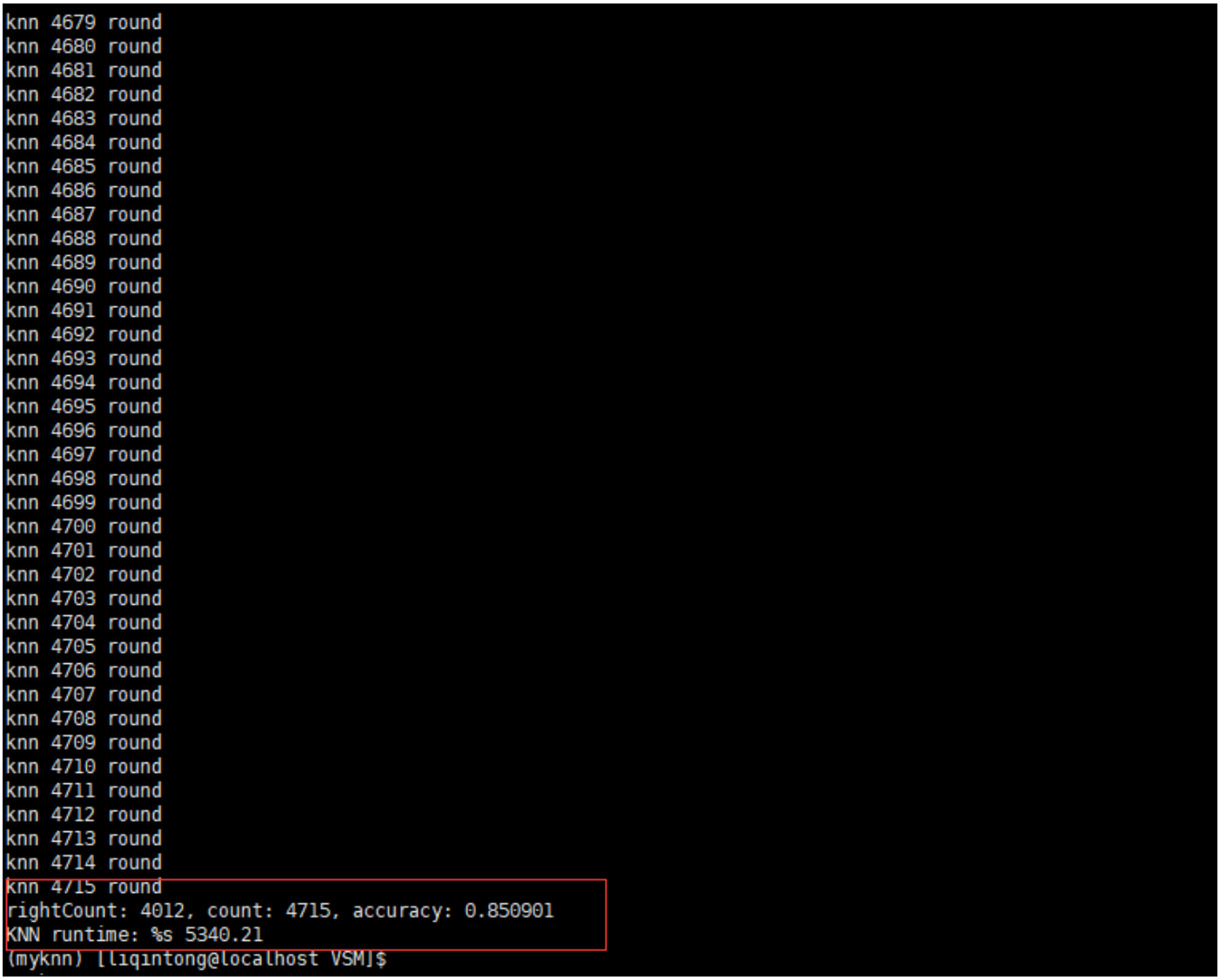
```
        classifyRight = item[0].split('_')[0]
        classifyResult = classifyResult[0][0]
        KNNResultWriter.write('%s %s\n' % (classifyRight, classifyResult))

        if classifyRight == classifyResult:
            rightCount += 1

    accuracy = float(rightCount) / float(count)
    print('rightCount: %d, count: %d, accuracy: %.6f' % (rightCount, count, accuracy))
```

5. 代码最终运行结果

准确率为 85%



四、 感想

本次 KNN 分类实验从构建词典开始，到最后分类文本，把课堂的知识实践了一遍，对算法更加熟悉，更全面地理解了数据挖掘知识，实验过程解决了很多 bug，对 python 语言有了更深入的了解，正所谓“纸上得来终觉浅，绝知此事要躬行”。

实验（二）

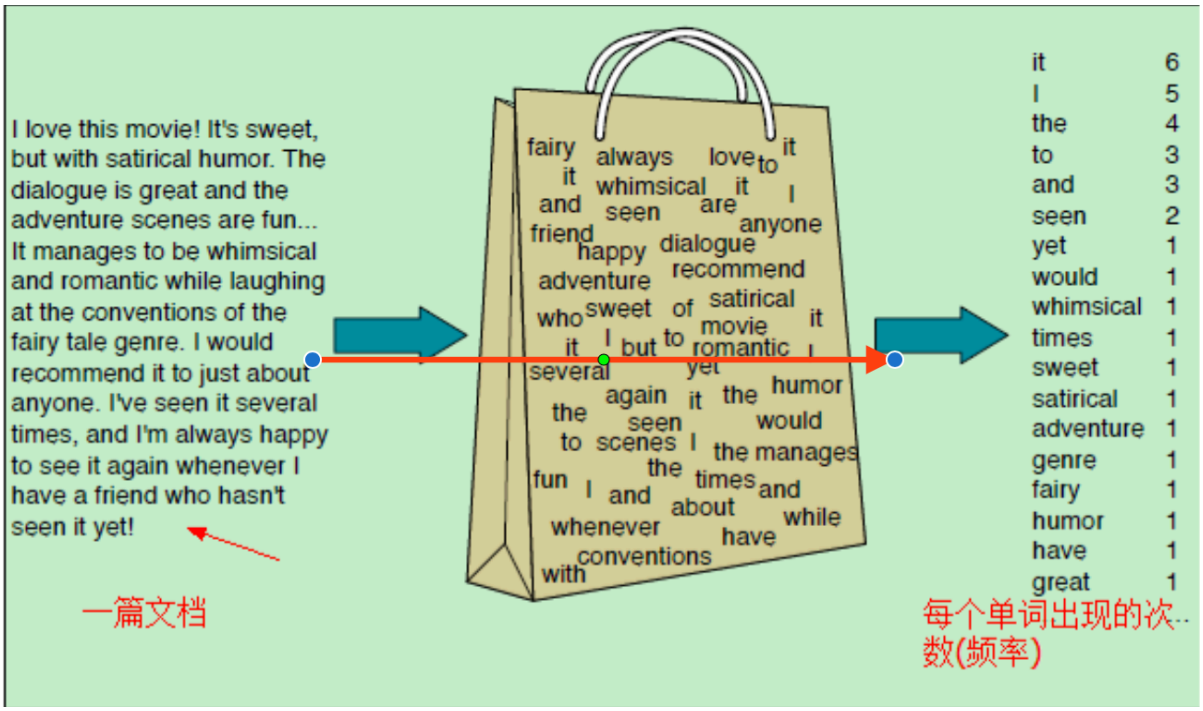
1. 实现朴素贝叶斯分类器，测试其在20 Newsgroups数据集上的效果
2. 20 Newsgroups数据集是大约20,000个新闻组文档的集合，在20个不同的新闻组中均匀分布

一、 分类目标

寻找文本的某些特征，根据这些特征将文本归为某个类，使用监督式机器学习将文本分类：假设已经有分好类的N篇文档，寻找一个分类器，当输入一个新文档d，输出d最可能属于的类别。

二、 词袋模型

词袋模型表示文本特征，给定一篇文档，有很多特征，比如文档中每个单词出现的次数、某些单词出现的位置、单词的长度、单词出现的频率。但是词袋模型只考虑一篇文档中出现的频率，用频率作为文档的特征。



三、 朴素贝叶斯分类器

1. 贝叶斯定理

$$P(c|x) = \frac{P(c)P(x|c)}{P(x)} = \frac{P(x,c)}{P(x)}$$

2. 朴素贝叶斯

朴素贝叶斯(Navie Bayes)是贝叶斯决策理论的一部分,只考虑最简单的假设，用 Python 将文本切分为词向量，然后利用词向量对文档分类。

优点：在数据较少的情况下仍然有效,可以处理多类别问题。

缺点：对于输入数据的准备方式较为敏感。

朴素贝叶斯分类器是一个概率分类器，假设现有类别C = {c1, c2, ⋯, cm}，给定文档d，文档d最可能属于的类：

$$C' = argmax\ P(c|d), \qquad c \in C$$

C'是条件概率最大的类，使用贝叶斯公式转换：

$$C' = argmaxP(c|d) = argmax\ \frac{P(d|c)P(c)}{P(d)}, \qquad c \in C$$

可以忽略分母，前面的$P(d|c)$是似然函数，后面的$P(c)$是先验概率

文档d的特征表示为d = {f1, f2, ⋯, fm}，fi实际是单词wi出现的次数。

$$C' = argmaxP(f1, f2, \dots, fn|c)P(c), \qquad c \in C$$

对文档d做假设，假设各个特征之间是相互独立的，那么:

$$P(f1, f2, \dots, fn|c) = P(f1|c) * P(f2|c) * \dots * P(fn|c), \qquad c \in C$$

由于每个概率值很小，若干个很小的概率值直接相乘，结果会越来越小，为了避免结果出现下溢，引入对数函数log，在log space中计算：

$$\mathrm{Cn} = argmax logP(c) + \sum_{i \in positions} logP(wi|c), \qquad \mathrm{c} \in \mathrm{C}$$

3. 代码实现

预处理：

句子内容的清理：去掉数字标点和非字母字符，去停用词；创建文件夹，存放预处理后的文本数据；建立目标文件夹，生成新文件，存放每个文档中的tokens；

构造字典：

统计每个词出现的次数，返回字典，按key排序，并打印字典；

选择特征词：

再生成20个文件夹，每个文件夹的每篇文档中存放本篇中的特征词；

创建训练集和测试集：

生成标注数据，i次实验的测试集，记录<doc, rightCate>个数，设置训练集和测试集的分割比例；

bayes对测试文档分类

```
# 11. bayes : p(c|d) = p(d|c)p(c)/p(d) = p(f1,f2,..|c)p(c)/p(d) = p(f1|c)p(f2|c)...p(fn|c)p(c)/p(d)
# traindir 类别m
# testFileWords 测试文档（特征词）
# cateWordNums 训练集类别，特征词总数 <cate, num>
# trainWordNum  训练集特征词总数
# CateWordsProbs 训练集类别m下，特征词c出现次数  <cate_m_word, num>

# p(f1|c) = (m类中f1出现次数 + 0.0001)/ cateWordNums_m + trainWordNum
# p(c)/p(d) = cateWordNums_m/ trainWordNum
```

计算准确率

四、 实验结果

```
训练集的单词总数: 2463481
cate 0 has 141129
cate 1 has 105670
cate 2 has 217339
cate 3 has 97530
cate 4 has 125767
cate 5 has 166871
cate 6 has 86529
cate 7 has 154479
cate 8 has 93955
cate 9 has 138140
cate 10 has 96778
cate 11 has 137428
cate 12 has 143494
cate 13 has 93036
cate 14 has 108327
cate 15 has 80296
cate 16 has 120168
cate 17 has 133739
cate 18 has 68200
cate 19 has 149550
cate_word_num: 230488
训练集的单词总数: 2458425
```

```
[liqintong@localhost NBC]$ python NBC_assignment2.py
1. Create Data
wordMap size: 117961
newWordMap size: 51700
print wordMap
wordMap size: 117961
newWordMap size: 51700
sortedWordMap size: 51700
wordMap size: 117961
newWordMap size: 51700
```

```
3. compute accuracy
naive bayes classifiers accuracy is: 0.875637
```

五、 感想

本次朴素贝叶斯文档分类实验从文档预处理开始，到最后对测试文档分类，把课堂的知识实践了一遍，对算法更加熟悉，更全面地理解了数据挖掘知识，实验过程解决了很多 bug，对 python 语言有了更深入的了解，正所谓“纸上得来终觉浅，绝知此事要躬行”。

实验（三）

1. 测试sklearn中以下聚类算法在tweets数据集上的聚类效果。
2. 使用NMI(Normalized Mutual Information)作为评价指标。

Method name	Parameters	Scalability	Usecase	Geometry (metric used)
K-Means	number of clusters	Very large `n_samples`, medium `n_clusters` with MiniBatch code	General-purpose, even cluster size, flat geometry, not too many clusters	Distances between points
Affinity propagation	damping, sample preference	Not scalable with n_samples	Many clusters, uneven cluster size, non-flat geometry	Graph distance (e.g. nearest-neighbor graph)
Mean-shift	bandwidth	Not scalable with `n_samples`	Many clusters, uneven cluster size, non-flat geometry	Distances between points
Spectral clustering	number of clusters	Medium `n_samples`, small `n_clusters`	Few clusters, even cluster size, non-flat geometry	Graph distance (e.g. nearest-neighbor graph)
Ward hierarchical clustering	number of clusters	Large `n_samples` and `n_clusters`	Many clusters, possibly connectivity constraints	Distances between points
Agglomerative clustering	number of clusters, linkage type, distance	Large `n_samples` and `n_clusters`	Many clusters, possibly connectivity constraints, non Euclidean distances	Any pairwise distance
DBSCAN	neighborhood size	Very large `n_samples`, medium `n_clusters`	Non-flat geometry, uneven cluster sizes	Distances between nearest points
Gaussian mixtures	many	Not scalable	Flat geometry, good for density estimation	Mahalanobis distances to centers
Birch	branching factor, threshold, optional global clusterer.	Large `n_clusters` and `n_samples`	Large dataset, outlier removal, data reduction.	Euclidean distance between points

一、 Normalized_mutual_into_score(NMI)

NMI ~ 两个聚类之间的归一化互信息

NMI 是 MI 分数的归一化，将结果在 0（无互信息）和 1（完全相关）之间进行缩放，在函数中，通过由 average_method 定义的 H(labels_true)和 H (labels_pred)的一些广义均值来标准化互信息。

已知聚类标签和真实标签，MI 可以测量两种标签排列之间的相关性，同时忽略标签中的排列，NMI 是对称函数，交换变量不改变函数值，可以做一致性检验，

假设对于 N 个样本，存在两种标签分配策略，分别记做 U，V。信息熵是一种信息不确定性的测度，定义如下：

$$H(U) = \sum_{i=1}^{|U|} P(i) \log P(i)$$

p(i) = |Ui|/N，即从 U 中随机选取一个样本属于类别 Ui 的概率

$$H(V) = \sum_{j=1}^{|V|} P'(j) \log P'(j)$$

p(j) = |Vj|/N，U 和 V 的互信息计算公式如下：

$$MI(U,V) = \sum_{i=1}^{|U|} \sum_{j=1}^{|V|} P(i,j) \log\left(\frac{P(i,j)}{P(i)P'(j)}\right)$$

NMI 定义如下：

$$NMI(U,V) = \frac{MI(U,V)}{\sqrt{H(U)H(V)}}$$

label_true: 将数据聚类为不相交的子集
label_pred: 将数据聚类为不相交的子集

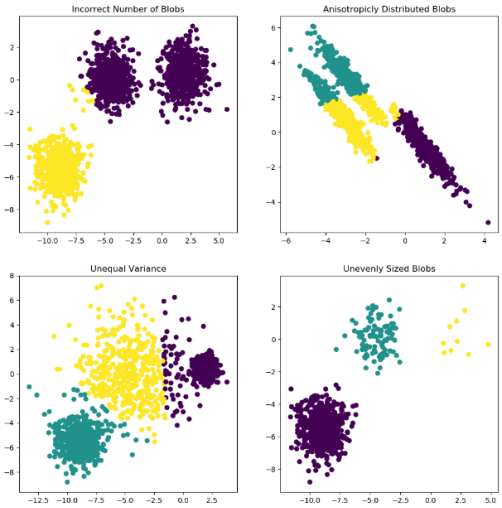
二、 K-Means

KMeans算法通过在n组相等方差中分离样本来聚类数据，最小化簇内平方和。该算法需要指定簇的数量，可以很好的扩展到大量样本，并且在许多不同领域应用。
KMeans算法将一组样本分成不相交的簇，每个簇由簇内样本均值描述，即簇内“质心”，KMeans旨在选择能够最小化平方差的质心。

$$\sum_{i=0}^{n} \min_{\mu_j \in C}(||x_i - \mu_j||^2)$$

簇内平方差（Inertia）是簇内紧密程度的度量，它也有缺点：

1. Inertia假设聚类是凸的和各向同性的，它对细长簇或不规则形状簇表现不佳
2. Inertia不是标准化的度量标准，值越小越好，零是最佳值，但是在高维空间，欧几里得距离会膨胀，即“维数诅咒”。在KMeans之前运行PCA等降维算法可以缓解该问题。



KMeans算法有三步：

1. 选择初始质心，最基本的方法是从数据集中选择样本
2. 将每个样本分配到最近的质心,计算簇内所有样本的平均值来创建新的质心
3. 计算旧质心和新质心之间的差异，并且重复2，3步，直到该值小于阈值

KMeans算法等价于小的，全相等的对角协方差矩阵期望最大化算法
结果如下：

```
KMeans NMI:  0.7629953372
```

三、 Affinity Propagation

AP 算法的基本思想是将全部样本看做网络的节点，然后通过网络中各条边的消息传递计算出个样本的聚类中心。聚类过程中，共有两种消息在各节点间传递，分别是吸引度（responsibility）和归属度（availability）。AP 算法通过迭代过程不断更新每一个点的吸引度和归属度，直到产生 m 个高质量的 Exemplar（相当于质心），同时将其余的数据点分配到相应的聚类中。

在样本之间发送消息来创建聚类，直到收敛。使用少量样本描述数据集，这些样本被识别为最具代表性的其他样本。成对之间发送的消息表示一个样本作为另一个样本的示例适合性，其响应于来自其他对的值而更新，这种更新迭代地发生直到收敛，此时选择最终样本，因此给出最终聚类。
缺点是复杂性，该算法具有顺序的时间复杂度，是样本的数量和直到收敛的迭代次数。如果使用密集相似性矩阵，则内存复杂度是有序的；如果使用稀疏相似性矩阵可以减少复杂度，AP 更适合中小型数据集。

在实际的使用中，AP 有两个需要手动设置的重要参数，preference 和 damping factor。前者定了聚类数量的多少，值越大聚类数量越多；后者控制算法的收敛效果。

```
AffinityPropagation NMI:  0.775145369374
```

四、 Mean-shift

Mean-Shift聚类法就可以自动确定k的个数，下面简要介绍一下其算法流程：

1. 随机确定样本空间内一个半径确定的高维球及其球心；
2. 求该高维球内质心，并将高维球的球心移动至该质心处；
3. 重复 2，直到高维球内的密度随着继续的球心滑动变化低于设定的阈值，算法结束

```
MeanShift NMI:  0.7056324482
```

五、 Spectral Clustering

谱聚类在样本之间生成低维度嵌入亲和度矩阵，然后在低维度空间中进行 KMeans，如果亲和度矩阵是稀疏的并且安装了 pyamg 模块，则尤其有效。SpectralClustering 需要指定簇的数量，适用于少数群集，但在使用多个群集时不建议使用。

谱聚类是基于图论的聚类方法，通过对样本数据的拉普拉斯矩阵的特征向量进行聚类。即把所有的数据看做空间中的点，这些点之间可以用边连接起来。距离较远的两个点之间的边权重值较低，而距离较近的两个点之间的边权重值较高，通过对所有数据点组成的图进行切图，让切图后不同的子图间边权重和尽可能的低，而子图内的边权重和尽可能的高，从而达到聚类的目

的。
对于两个聚类，它解决了相似性图上的归一化切割问题的凸松弛：将图切成两半，使得切割边缘的权重与每个聚类内边缘的权重相比较小。

```
SpectralClustering NMI:  0.47384412442
```

六、 Ward hierarchical clustering

Hierarchical clustering（分层聚类）通过连续合并或拆分嵌套聚类来构建嵌套聚类，这种簇的层次结构表示为树，树根是收集所有样本的唯一聚类，叶子是只有一个样本的聚类。

```
Ward Hierarchical clustering NMI:  0.759773200943
```

七、 Agglomerative Clustering

Agglomerative Clustering 使用自下而上的方法执行层次聚类：每个观察在其自己的集群中开始，并且集群被连续地合并在一起。链接标准确定用于合并策略的度量标准：

- Ward 最小化所有聚类中的平方差的总和。它是一种方差最小化方法，在这个意义上类似于 k 均值目标函数，但采用凝聚分层方法。
- Maximum or complete linkage（最大或完整的连接）最小化了簇对的观察之间的最大距离。
- Average linkage（平均连锁）最小化了所有簇对的观察之间的距离的平均值。
- Single linkage（单个连锁）最小化最近的簇对观察之间的距离。

当与连接一起使用时，Agglomerative Clustering 还可以扩展到大量样本

```
AgglomerativeClustering NMI:  0.759773200943
```

八、 DBSCAN

DBSCAN 算法将簇视为由低密度区域分隔的高密度区域，DBSCAN 发现的簇可以是任何形状，而 KMeans 假设聚类是凸形的，DBSCAN 的核心组件是核心样本，是高密度区域的样本，因此簇是一组核心样本，每个核心样本彼此接近，一组接近核心样本的非核心样本。
DBSCAN 类的重要参数也分为两类，一类是 DBSCAN 算法本身的参数，一类是最近邻度量的参数，下面我们对这些参数做一个总结。
1. eps： ϵ-邻域的距离阈值，与样本距离超过ϵ的样本点不在ϵ-邻域内。默认值是 0.5。eps 过大，更多的点会落在核心对象的ϵ-邻域，此时类别数可能会减少， 本来不应该是一类的样本也会被划为一类。反之则类别数可能会增大，本来是一类的样本却被划分开。
2. min_samples： 样本点要成为核心对象所需要的ϵ-邻域的样本数阈值。默认值是 5。通常和 eps 一起调参。在 eps 一定的情况下，min_samples 过大，则核心对象会过少，此时簇内部分本来是一类的样本可能会被标为噪音点，类别数也会变多。反之 min_samples 过小的话，则会产生大量的核心对象，可能会导致类别数过少。
3. metric：最近邻距离度量参数。
4. algorithm：最近邻搜索算法参数，有 4 种可选输入，‘brute’：蛮力实现，‘kd_tree’：KD 树实现，‘ball_tree’：球树实现， ‘auto’在上面三种算法中做权衡，选择一个拟合最好的最优算法。如果输入样本特征是稀疏的时候，最后 scikit-learn 都会去用蛮力实现‘brute’。
5. leaf_size：最近邻搜索算法参数。
6. p: 最近邻距离度量参数。p=1 为曼哈顿距离， p=2 为欧式距离。

```
DBSCAN NMI:  0.155256389516
```

九、 Gaussian Mixture

sklearn.mixture 是一个包，使人们能够学习高斯混合模型（支持对角线，球形，并列和完全协方差矩阵），对它们进行采样，并根据数据进行估计。 还提供了帮助确定适当数量的组件的设施。

高斯混合模型是概率模型，其假设所有数据点是从具有未知参数的有限数量的高斯分布的混合生成的。 可以将混合模型视为概括 k 均值聚类，以合并关于数据的协方差结构以及潜在高斯的中心的信息。

Scikit-learn 实现了不同的类来估计高斯混合模型，这些模型对应于不同的估计策略，详述如下。

```
Gaussian mixture models NMI:  0.800481598098
```

十、 Birch

Birch 为给定数据构建一个成为 Characteristic Feature Tree（CFT）的树，数据基本上被有损压缩为一组 CF 节点，CF 节点有许多 CF 子集群，位于非终端 CF 节点中的 CF 子集可以将 CF 节点作为子节点。

CF 子集包含集群所需的信息，可以防止整个输入数据保存在内存中，这些信息包括:

- 子集群中的样本数
- 线性求和：保持所有样本总和的 n 维向量
- 平方和：所有样本的平方 F2 范数之和
- 质心：避免重新计算 n 个样本的线性和
- 质心的平方标准

```
Birch NMI:  0.780857693264
```

十一、 总结

TF-IDF：

使用 TF-IDF 特征表示文档，TF 是词频，统计文本中各个词的出现频率，并作为文本特征；IDF 反应了一个词在所有文本中出现的频率，如果一个词在很多的文本中出现，那么它的 IDF 值应该低，而反过来如果一个词在比较少的文本中出现，那么它的 IDF 值应该高，如果一个词在所有的文本中都出现，那么它的 IDF 值应该为 0。

TF-IDF 可以使用 Sklearn 包的 CountVectorizer 类向量化之后再调用 TfidfTransformer 类进行预处理。第二种方法是直接用 TfidfVectorizer 完成向量化与 TF-IDF 预处理。

将 TF-IDF 表示的文档作为输入，使用 sklearn 聚类函数进行聚类。

skearn 不同聚类特点比较：

K-means 需要设置类别的数目，可扩展性比较强，但是由于计算原理比较简单（只是计算点之间的距离），所以只能应用于一些比较 general 的分类目的，适合于每个类别的数量差不多，并且不能有太多的类别的数据集。

Affinity propagation 会根据数据自主选择类别的数量，计算的是图距离（非平面几何距离），可以有很多类别而且类别的数目可以不均匀，但是它的计算时间空间复杂度都很高，比较适合于小数据。

Mean-shift 计算的是平面几何距离，可以有很多类别而且类别的数目可以不均匀。

Spectral clustering 需要给定类别数量，适合于少类别的数据集，类别量可以不均匀，它计算的是图距离。

Ward hierarchical clustering 需要给定类别数量，可以用于大数据集，即类别数量多，类别样本多，计算的是几何距离。

Agglomerative clustering 需要给定类别数量，可以用于大数据集，即类别数量多，类别样本多，计算的非欧几里德距离。

DBSCAN 需要给定相邻样本定的距离，可以适合用于数据量超大的数据集，类别量可以不平衡。

Gaussian mixtures 有很多需要定义的参数，有利于密度计算。

Birch 可以用于大数据集，有异常值删除，计算的是点之间的欧几里德距离。

skearn 不同聚类示例可视化比较：