Information Theory Project 1 Report

周　游　118039910146
张靖宜　118039910143
杨　航　018034910063
于泽汉　118039910141

1 Tasks and requirements

Please use the dataset "AAAI 14 Accepted Papers" to do the following tasks:

1. Apply the relative entropy rule to cluster different papers into different groups according to their titles, abstracts, keywords.
2. Besides the relative entropy rule, please apply another method to achieve the target in 1.
3. Please write a report according to your experiments. Note that in your report, you need to include the following contents:
   (a) The contributions of different members.
   (b) Algorithm analysis
   (c) Some important codes for your experiments.
   (d) The results of your experiments.

Note: the dataset "AAAI 14 Accepted Papers" consists of the titles, authors, topics, keywords and abstracts of all accepted papers in AAAI, 2014.

2 Basic infomation

2.1 Contributions of members

周　游　**Algorithms analysis**
Search useful and practical methods to cluster the documents.
Analyze and choose feasible algorithms and techniques.

张靖宜　**Programming**
Process the texts and data.
Put the ideas and methods into practice, and implement the algorithms.

杨　航　**Code testing**
Test the codes and improve the algorithms.
Control the workflow and help team communication.

于泽汉　**Report writing**
Determine the workflow of this project and devide the work to team members.
Collect members' work and write the report.

2.2 Environments and tools

Operating system:	Windows 10 Pro
Programming language:	Python
Writing software:	LaTeX

3 Ideas, methods, techniques, codes and results

3.1 Preprocessing

Since the given dataset is a .csv file, we need to preprocess it and translate to some formats friendly to computers.

3.1.1 Tokenize texts

The given dataset contains titles, authors, topics, keywords and abstracts. As we read these texts as 'string' format in Python, we should extract the words from the strings, or more precisely, tokenize the texts.

The following codes show how we tokenize the texts and split them to lists of words:

Listing 1: Tokenize texts

```
import re
import nltk
from nltk.tokenize import word_tokenize, sent_tokenize
from nltk.stem import WordNetLemmatizer
import pandas as pd

# title, authors, groups, keywords, topics, abstract
df = pd.read_csv('./aaai14.csv', sep=',', encoding='utf8')
# header_list = ['title', 'keywords'] #, 'abstract']
header_list = ['title', 'keywords', 'abstract']
docnum = len(df)

def tokenizeText(text):
    # s = "string. With. Punctuation?"
    text = re.sub(r'[^\w\s]', '', text)
    term_list = word_tokenize(text)
    return term_list
```

3.1.2 Lemmatize texts

Although we have cut the strings to words, these words still needs to be processed further.

For example, in most cases of our dataset, the noun 'Algorithms' is (almost) the same as 'algorithm', and the verb 'learn' represents the same meaning as 'learning' or 'learnt'.

So we should convert the plurals to singulars, make the verb tenses consistent and some other similar word conversions. This process is the so-called 'lemmatizing texts'.

The following codes show how we lemmatize the texts.

Listing 2: Lemmatize texts

```
wnl = WordNetLemmatizer()
def lemmatizeTerm(term):
    term_out = term
    term_out = wnl.lemmatize(term_out, pos='n').lower()
    term_out = wnl.lemmatize(term_out, pos='v').lower()
    return term_out

def lemmatizeText(text):
    if   isinstance(text, str):
```

```
        text_out = lemmatizeTerm(text)
    elif isinstance(text, list):
        text_out = []
        for term in text:
            text_out.append(lemmatizeTerm(term))
    return text_out
```

3.1.3 Get all the distinct terms

After tokenizing and lemmatizing the texts, we can then get all the distinct terms.

This is implemented with the following codes:

Listing 3: Get all items

```
def getAllTerm():
    for i in range(docnum):
        for header in header_list:
            # print('>>> Processing doc {:0>4} ... '.format(i))
            text = df[header][i]
            term_list = lemmatizeText(tokenizeText(text))
            for term in term_list:
                if term in all_term:
                    pass
                else:
                    all_term.append(term)
getAllTerm()
with open('all_term.txt', 'w', encoding='utf8') as all_term_file:
    print(*all_term, sep='\n', file=all_term_file)
```

Through the process above, we can get all distinct terms. In our case, there are 5235 distinct terms. The content of `all_term.txt` is like this:

Listing 4: all_term.txt

```
kernelized
bayesian
transfer
learn
crossdomain
domain
adaptation
kernel
...
...
...
rival
spite
stsc
```

3.2 Calculate TF-IDF

3.2.1 Definitions of important variables

After preprocessing the dataset, we can calculate some important variables to prepair for the clustering.

The definition of these variables are shown below:

$\boldsymbol{D}$:	The set of documents
$\boldsymbol{T}$:	The set of distinct terms occurring in all documents
$\boldsymbol{tf(d,t)}$:	The frequency of term t in document d
$\boldsymbol{df(t)}$:	The number of documents containing term t
$\boldsymbol{tf\text{-}idf(d,t)}$:	The term frequency - inverse document frequency

3.2.2 D and T

We have got D and T in the previous section.

The information D is saved in a .csv file, and after preprocessing (tokenizing and lemmatizing), we get T.

3.2.3 $tf(d,t)$

tf is a $|D| \times |T|$ matrix, which can be calculated by the following codes:

Listing 5: Calculate tf matrix

```
tf_matrix = [[0] * len(all_term) for _ in range(docnum)]
def calcTF():
    # tf: term frequency
    # return a 2d array
    for i in range(docnum):
        for header in header_list:
            text = df[header][i]
            term_list = lemmatizeText(tokenizeText(text))
            for term in term_list:
                tf_matrix[i][all_term.index(term)] += 1
print('Calculating tf_matrix ...')
calcTF()
with open('tf_matrix.txt', 'w') as tf_matrix_file:
    for row in tf_matrix:
        print(*row, sep=' ', file=tf_matrix_file)
```

In our case, the size of tf matrix is 398×5235. The content of the `tf_matrix.txt` is like this:

Listing 6: tf_matrix.txt

```
2 2 5 6 1 3 2 1 2 2 2 1 1 1 2 5 2 5 1 5 1 6 1 ...
0 0 6 8 0 3 0 0 1 0 0 0 0 0 0 3 0 3 0 4 0 5 1 ...
0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 2 0 0 3 0 3 0 ...
0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 0 1 0 2 0 2 0 ...
0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 2 0 2 0 7 0 ...
0 0 0 3 0 0 0 0 3 0 0 2 0 0 0 0 0 2 0 4 0 3 0 ...
0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 2 0 4 0 6 0 ...
...
...
...
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 3 2 0 7 0 7 0 ...
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 3 0 0 0 4 0 ...
0 0 6 4 0 8 0 0 2 0 0 0 0 1 0 1 0 1 0 7 0 2 0 ...
```

3.2.4 $df(t)$

df is a $1 \times |T|$ matrix, which can be calculated by the following codes:

Listing 7: Calculate df matrix

```
df_matrix = [0] * len(all_term)
def calcDF():
    # df: num of docs containing the term 't'
    # return a 1d array
    for col in range(len(tf_matrix[0])):
        for row in range(len(tf_matrix)):
            if tf_matrix[row][col] == 0:
                pass
            else:
                df_matrix[col] += 1
print('Calculating df_matrix ...')
calcDF()
with open('df_matrix.txt', 'w') as df_matrix_file:
    print(*df_matrix, sep='\n', file=df_matrix_file)
```

In our case, the size of df matrix is 1×5235. The content of the `df_matrix.txt` is like this:

Listing 8: df_matrix.txt

```
2 20 15 159 5 70 8 8 ... 1 1 1
```

3.2.5 $tf\text{-}idf(d,t)$

$tf\text{-}idf$ is also a $|D| \times |T|$ matrix, which is one of the most important variable in our algorithm, shows the importance of term t in document d.

The formula is:

$$tf\text{-}idf(d,t) = tf(d,t) \times \log \frac{|D|}{df(t)}$$

We take e as the base of the logarithm.

It can be calculated by the following codes:

Listing 9: Calculate tf-idf matrix

```
tf_idf_matrix = [[0] * len(all_term) for _ in range(docnum)]
def calcTFIDF():
    # tf: term frequency - inverse document frequency
    # return a 2d array
    for row in range(len(tf_idf_matrix)):
        for col in range(len(tf_idf_matrix[0])):
            tf_idf_matrix[row][col] = round(tf_matrix[row][col] * log(docnum / df_matrix[
                col]), 3)

print('Calculating tfidf_matrix ...')
calcTFIDF()
with open('tf_idf_matrix.txt', 'w') as tf_idf_matrix_file:
    for row in tf_idf_matrix:
        print(*row, sep=' ', file=tf_idf_matrix_file)
```

In our case, the size of tf-idf matrix is 398×5235. The content of the `tf_idf_matrix.txt` is like this:

Listing 10: tf_idf_matrix.txt

```
10.587 5.981 16.392 5.505 4.377 5.214 7.814 ...
0.0 0.0 19.67 7.34 0.0 5.214 0.0 0.0 0.949  ...
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.949 0.0 0 ...
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.949 0.0 0 ...
0.0 0.0 0.0 2.753 0.0 0.0 0.0 0.0 2.848 0.0 ...
0.0 0.0 0.0 0.918 0.0 0.0 0.0 0.0 0.949 0.0 ...
...
...
...
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 ...
0.0 0.0 19.67 3.67 0.0 13.904 0.0 0.0 1.899 ...
```

3.3 Calculate original and averaged relative entropy (K-L divergence)

Afeter getting the *tf-idf* matrix, we can then calculate the 'distance' of different documents, which is used to measure the differences among documents. We take the relative entropy as the measurement of 'distance' here. (We will choose another variable of 'distance' later.)

The definition of $\boldsymbol{kld(a,b)}$ is: The relative entropy (Kullback-Leibler divergence) from b to a.

kld is a $|D| \times |D|$ matrix. The formula is:

$$kld(a,b) = D_{KL}\left(\overrightarrow{a} \middle\| \overrightarrow{b}\right) = \sum_{t=1}^{t=|T|} tf\text{-}idf(a,t) \times \log \frac{tf\text{-}idf(a,t)}{tf\text{-}idf(b,t)}$$

It can be calculated by the following codes:

Listing 11: Calculate K-L divergence matrix

```
def calcKLD(vec1, vec2):
    # kld: KL divergence
    kld = 0
    for i in range(len(vec1)):
        if vec1[i]==0 or vec2[i]==0:
            pass
        else:
            kld += vec1[i] * log(vec1[i] / vec2[i])
    return kld
```

While the original K-L divergence is not symmetric, we need to define another variable, which is called averaged K-L divergence here. It is defined as:

$$akld(a,b) = D_{AKL}\left(\overrightarrow{a} \middle\| \overrightarrow{b}\right) = \frac{1}{2} \times \left[D_{KL}\left(\overrightarrow{a} \middle\| \overrightarrow{b}\right) + D_{KL}\left(\overrightarrow{b} \middle\| \overrightarrow{a}\right)\right] = \frac{1}{2} \times [kld(a,b) + kld(b,a)]$$

It can be calculated by the following codes:

Listing 12: Calculate averaged K-L divergence matrix

```
kld_matrix  = [[0] * docnum for _ in range(docnum)]
akld_matrix = [[0] * docnum for _ in range(docnum)]

def calcAKLD():
    # akld: average KL divergence
    for row in range(docnum):
        for col in range(docnum):
            kld_matrix[row][col] = calcKLD(tf_idf_matrix[row], tf_idf_matrix[col])

    for row in range(docnum):
        for col in range(docnum):
            akld_matrix[row][col] = round(1/2 * (kld_matrix[row][col] + kld_matrix[col][
                row]), 3)
print('Calculating alkd_matrix ...')
calcAKLD()
with open('akld_matrix.txt', 'w') as alkd_matrix_file:
    for row in akld_matrix:
        print(*row, sep=' ', file=alkd_matrix_file)
```

In our case, the size of averaged K-L divergence matrix is 398×398. The content of the `akld_matrix.txt` is like this:

Listing 13: akld_matrix.txt

```
0.0 61.046 4.543 4.082 4.192 26.149 17.33 14.538 ...
61.046 0.0 0.48 2.087 2.426 22.007 22.522 14.004 ...
4.543 0.48 0.0 5.614 3.179 2.667 7.793 12.05 ...
4.082 2.087 5.614 0.0 2.514 3.682 7.678 15.796 ...
4.192 2.426 3.179 2.514 0.0 11.187 7.802 7.536 ...
...
...
...
11.937 4.136 4.019 7.831 1.77 5.305 6.201 3.552 ...
7.416 3.767 0.806 3.579 1.611 0.504 3.074 5.035 ...
46.382 46.474 1.041 9.891 3.004 14.849 10.183 9.678 ...
```

3.4 Clustering with k-means-like (not typical k-means) algorithm

There are many practical methods to cluster the documents. Since k-means algorithm is simple and fast, we take it as our clustering algorithm.

The key steps of k-means algorithm are:

1. **Initialization**: Choose k random points and take each as a group, the centroid of each group is the coordinate of the initial point.
2. **Assignment/Expectation**: Choose new points, calculate the 'distance' to the centroid of each group, and put the new points into the group whose centroid is 'closest' to the new point.
3. **Update/Maximization**: Update the centroid of each group.
4. **Iteration**: Repeat step 2 and 3, until convergence has been reached.

However, what we have is just the KL divergence matrix, which only records distances of different documents, instead of coordinates, so we cannot simply define the 'centroid' in this case. Thus, we need to modify the k-means algorithm, in which case we can still benifit from the core ideas of k-means algorithm.

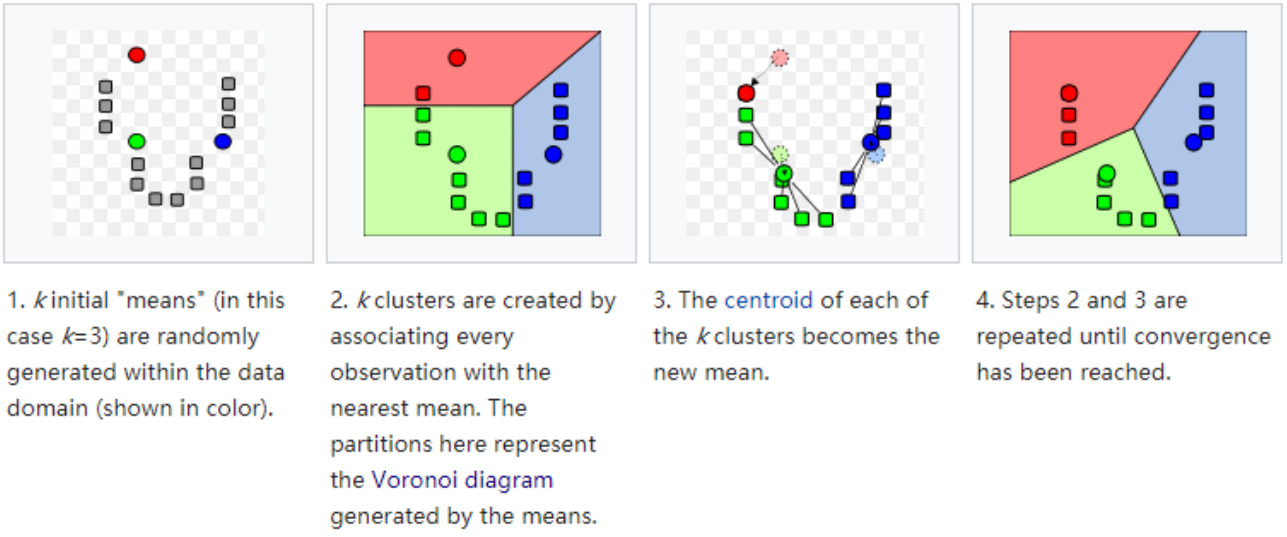

Figure 1: The typical k-means algorithm

The modificaiton contains several aspects:

1. Do not choose initial points randomly, but choose the points among which the distances are as far as possible.
2. Forsake the concept of 'centroid' and calculate the average distances from the new point to all points in the same group each time.
3. Do not keep iterating until convergence is reached, but traverse all the points once.

So our k-means-like (not k-means) algorithm is shown as follows:

1. **Initialization**: Choose k random document and take each as a group.
2. **Assignment**: Choose another new document, for each group, sum the averaged K-L divergence ('distance') from the new document to all elements in this group, calculate the mean distance, and put the new document into the group whose mean distance is the smallest.
3. **Traversal**: Repeat step 2, until all documents are grouped.

The modules of the algorithm is implemented by the following codes:

Listing 14: Choose initial documents

```
from random import randint

groups = [[] for _ in range(k)]
grouped_doc = []
ungrouped_doc = [i for i in range(0, docnum)]

def chooseInitDoc():
    global groups, grouped_doc, ungrouped_doc
    groups = [[] for _ in range(k)]
    grouped_doc = []
    ungrouped_doc = [i for i in range(0, docnum)]

    first_doc = randint(0, docnum-1)
    groups[0].append(first_doc)

    ungrouped_doc.pop(first_doc)
    grouped_doc.append(first_doc)

    for i in range(1, k):
```

```
        distance_this = 0
        doc_tmp = 0
        idx_tmp = 0
        for new_idx, new_doc in enumerate(ungrouped_doc):
            distance_tmp = 0
            for old_doc in grouped_doc:
                distance_tmp += akld_matrix[new_doc][old_doc]

            if distance_tmp >= distance_this:
                distance_this = distance_tmp
                doc_tmp = new_doc
                idx_tmp = new_idx
        ungrouped_doc.pop(idx_tmp)
        grouped_doc.append(doc_tmp)
        groups[i].append(doc_tmp)
    with open('cluster_results.txt', 'a') as crf:
        print('initial doc id:', file=crf)
        print(*grouped_doc, sep=' ', file=crf)
        print('', file=crf)
```

Listing 15: Update groups

```
def updateGroups():
    global groups, grouped_doc, ungrouped_doc
    for i in range(len(ungrouped_doc)):
        new_idx = randint(0, len(ungrouped_doc)-1)
        new_doc = ungrouped_doc[new_idx]
        distance_this = float('inf')
        gid_tmp = 0
        for gid, group in enumerate(groups):
            distance_tmp = 0
            for gele in group:
                distance_tmp += akld_matrix[gele][new_doc]
            distance_tmp = distance_tmp / len(group)
            if distance_tmp <= distance_this:
                distance_this = distance_tmp
                gid_tmp = gid
        ungrouped_doc.pop(new_idx)
        groups[gid_tmp].append(new_doc)
        grouped_doc.append(new_doc)
```

Listing 16: Output information of final groups

```
id_list, sz_list = [], []
def outputGroups():
    global id_list, sz_list
    id_list, sz_list = [], []
    for gid, group in enumerate(groups):
        id_list.append(gid)
        sz_list.append(len(group))
        with open('cluster_results.txt', 'a') as crf:
            print(f'id: {gid}, size:{len(group)}', file=crf)
            print(*group, sep=' ', file=crf)
```

Since we do not know which k is suitable in our case, we loop k from 2 to 33 and see which one is good. This is implemented as the following:

Listing 17: Loop k and visualize the results of clustering

```
import matplotlib.pyplot as plt
import matplotlib.ticker

k_min, k_max = 2, 33
k_list = [i for i in range(k_min, k_max+1)]
plt.figure(figsize=(1200/72, 960/72), dpi=72)
for idx, k in enumerate(k_list):
    with open('cluster_results.txt', 'a') as crf:
        print(f'k = {k}')
        print(f'\nk = {k}', file=crf)
    col_num = 4
    row_num = int((k_max - k_min + 1) / col_num) + 1
    # col_id = (idx) % col_num + 1
    plt.subplot(row_num, col_num, idx+1)
    chooseInitDoc()
    updateGroups()
    outputGroups()
    plt.bar(id_list, sz_list)
    plt.gca().set_xlim([id_list[0]-0.5, id_list[-1]+0.5])
    tick_step = int((k-1)/10) + 1
    locator = matplotlib.ticker.MultipleLocator(tick_step)
    plt.gca().xaxis.set_major_locator(locator)
    formatter = matplotlib.ticker.StrMethodFormatter("{x:.0f}")
    plt.gca().xaxis.set_major_formatter(formatter)
    plt.xlabel(f'k = {k}')
plt.tight_layout()
plt.savefig('./images/cluster_results.png', dpi=72, bbox_inches='tight')
plt.show()
```

We observe the number of elements in each group, and check whether the clustering is good by looking into their distribution.

The results is shown as follows:

From the figure of distributions, we choose $k = 6$ as the number of groups, because the distribution in this case is more uniform than others. Maybe the uniformity of distributions is not really a good criterion to choose the appropriate k, we just take it.

Figure 2: Distributions of numbers of elements in groups (k from 2 to 33)

So the information of groups under $k = 6$ is shown like this:

Listing 18: Results of clustering when $k = 6$

```
k = 6

initial doc id:
7 70 8 268 245 85

[id: 0, size: 63]
7 356 117 395 355 372 316 392 259 185 141 71 17 32 113 171 114 60 240 29 161 130 64 385
62 376 98 35 137 390 78 303 384 281 38 322 284 387 183 286 287 116 67 200 180 91 364 96
76 138 1 57 261 179 118 269 339 56 295 189 191 306 352

[id: 1, size: 79]
70 195 273 109 258 206 50 210 174 23 321 278 128 367 346 250 353 226 124 292 363 9 69 127
341 307 283 276 45 134 242 296 243 254 221 110 27 272 360 345 131 146 173 324 224 204
265 190 129 266 10 39 311 197 143 89 86 136 205 74 232 228 319 354 0 19 122 193 334 111
42 172 44 84 315 95 93 391 271

[id: 2, size: 51]
8 326 159 11 325 274 46 277 217 312 72 160 361 94 119 291 229 2 88 216 170 298 53 175
238 157 323 81 358 103 58 244 15 263 394 106 213 163 108 30 47 165 51 368 253 215 320 196
3 328 236

[id: 3, size: 73]
268 301 302 275 349 186 192 378 381 288 181 350 199 362 300 92 102 383 148 256 347 396 4
24 12 18 317 297 104 333 280 121 73 176 225 255 120 299 230 252 335 207 13 262 145 201
107 382 211 314 43 336 219 370 123 33 309 68 83 340 75 168 66 132 247 65 330 142 251 31
331 264 289

[id: 4, size: 72]
245 270 257 304 285 329 374 187 223 25 153 150 156 308 147 115 342 140 365 379 388 54 209
100 233 231 377 167 77 249 133 208 239 220 22 237 164 282 162 48 203 359 212 144 293 87
126 248 375 279 125 318 371 5 351 234 366 198 397 380 80 105 260 214 202 386 97 343 227
218 155 55

[id: 5, size: 60]
85 166 152 305 310 158 357 344 14 313 182 61 21 63 177 267 294 20 338 235 348 332 40 59
90 149 82 241 99 6 101 178 327 79 49 369 151 184 169 393 41 188 139 26 290 28 389 34 37
246 222 52 112 16 36 135 373 194 154 337
```

The `id` is the identification of each group, the `size` is the number of documents in each group, and the number in each group is the index of document in the dataset. (Note that the identification of the group and the index of the document both start with 0 instead of 1.)

Since the first initial point is chosen randomly, the results of clustering may change a little after each rerun, but the documents in the same group will be roughly the same.

3.5 Clustering with another method: Spectral Clustering

In multivariate statistics and the clustering of data, spectral clustering techniques make use of the spectrum (eigenvalues) of the similarity matrix of the data to perform dimensionality reduction before clustering in fewer dimensions.

The similarity matrix is provided as an input and consists of a quantitative assessment of the relative similarity of each pair of points in the dataset.

Similar to the method above, we loop k from 2 to 33 and see which one is good.

We do not bother to reimplement the algorithm again, but use an existed framework named `sklearn`.

This is implemented as the following:

Listing 19: Spectral clustering

```
from sklearn.cluster import SpectralClustering
groups = [[] for _ in range(k)]
def spectralCluster(k):
    global groups
    groups = [[] for _ in range(k)]
    spectral_cluster = SpectralClustering(n_clusters=k).fit(tf_idf_matrix)
    for doc in range(len(spectral_cluster.labels_)):
        gid = spectral_cluster.labels_[doc]
        groups[gid].append(doc)
```

Similar to the analysis above, we observe the number of elements in each group, and check whether the clustering is good by looking into their distribution.

The results is shown as follows:

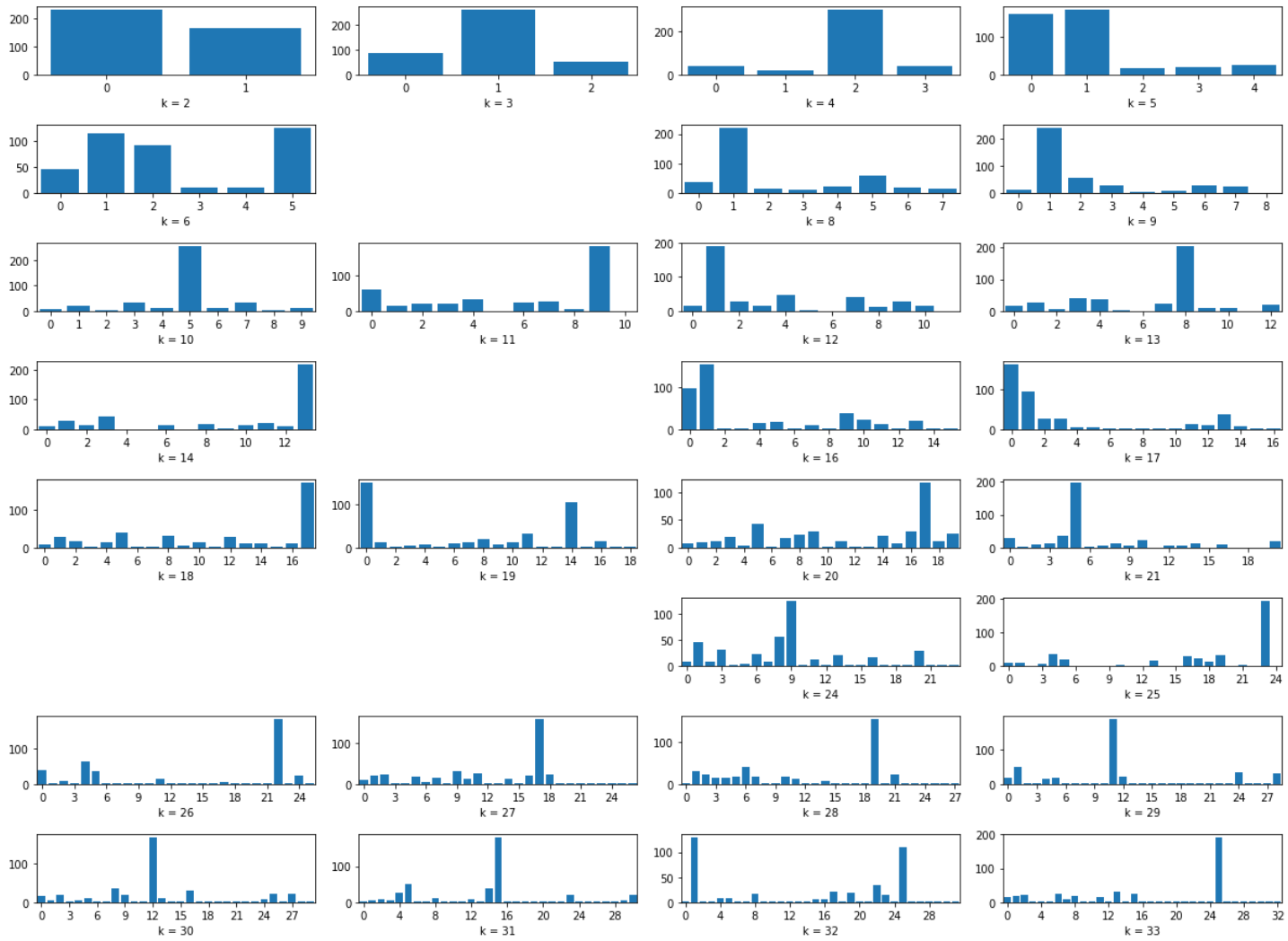

Figure 3: Distributions of numbers of elements in groups (k from 2 to 33) with spectral clustering algorithm

Not every k can be used as the number of groups in the spectral clustering, so some are missing in the figure.

Again, from the figure of distributions, we choose $k = 6$ as the number of groups because its distribution is more 'normal' than others.

The information of groups under $k = 6$ is shown like this:

Listing 20: Results of spectral clustering when $k = 6$

```
k = 6

[id: 0, size: 46]
25 27 41 51 56 61 70 80 82 89 90 93 102 116 119 131 132 150 153 163 171 174 180 198 199
204 205 229 245 256 258 263 269 273 278 280 308 329 330 332 338 345 354 358 365 367

[id: 1, size: 114]
0 1 2 3 4 6 7 16 23 30 36 37 38 40 42 45 48 52 65 66 67 72 73 76 81 87 88 91 97 100 103
110 113 115 121 123 124 125 129 130 133 134 137 145 148 155 165 166 167 170 175 178 179
183 188 191 192 197 203 206 209 213 217 228 231 234 235 238 240 241 248 250 254 259 260
262 266 283 284 285 286 292 295 299 306 314 316 319 320 321 323 326 334 337 340 343 346
348 351 352 356 360 361 364 379 380 381 382 384 387 393 394 395 397

[id: 2, size: 92]
5 9 11 12 17 18 19 21 22 24 29 47 55 58 62 78 79 96 98 99 104 111 117 120 126 127 128
136 140 141 146 147 151 158 159 162 168 172 173 177 181 182 186 187 189 194 200 214 215
218 219 220 222 225 232 242 243 249 252 253 255 275 277 279 282 287 288 291 293 294 297
301 302 309 318 327 328 331 333 336 341 349 350 357 359 368 371 374 383 385 386 391

[id: 3, size: 11]
71 196 208 223 237 268 296 317 325 342 388

[id: 4, size: 10]
34 39 83 84 85 139 154 160 272 370

[id: 5, size: 125]
8 10 13 14 15 20 26 28 31 32 33 35 43 44 46 49 50 53 54 57 59 60 63 64 68 69 74 75 77
86 92 94 95 101 105 106 107 108 109 112 114 118 122 135 138 142 143 144 149 152 156 157
161 164 169 176 184 185 190 193 195 201 202 207 210 211 212 216 221 224 226 227 230 233
236 239 244 246 247 251 257 261 264 265 267 270 271 274 276 281 289 290 298 300 303 304
305 307 310 311 312 313 315 322 324 335 339 344 347 353 355 362 363 366 369 372 373 375
376 377 378 389 390 392 396
```

Again, the `id` is the identification of each group, the `size` is the number of documents in each group, and the number in each group is the index of document in the dataset. (Note that the identification of the group and the index of the document both start with 0 instead of 1.)

4 Conclusions

From the results of two clustering algorithms, we find that the relative entropy (K-L divergence) is much useful in document clustering.

Another key variable in the process of clustering is the tf-idf matrix. Both algorithms are based on this matrix. We could also choose some other variables as the representations of the information which documents contain.

In general, the results are not bad, but if not for the reason that the size of the dataset is small, we could get better results and apply more methods and techniques.

We have leant much from this project, and it still leaves us a lot to improve.