Information Retrieval

1. Inverted index and Boolean Retrieval Model

实验内容

1. 使用数据集30548条tweets，读取其中的text中的数据，来创建 inverted index
2. 实现 Boolean Retrieval Model：
    - Input：a query (like Ron Weasley birthday)
    - Output: print a list of top k relevant tweets.
    - 支持and, or ,not；查询优化可以选做；

实验思路

倒排索引

1.读取tweets中的text中的数据,进行数据的预处理，主要是采用**正则表达式**进行分词和对一些符号进行处理，只保存得到字母和数字

```
def token_stream(line):
    # re.I 不区分大小写
    li=nltk.word_tokenize(line)
    li=' '.join(li)
    return re.findall(r'\w+', li,re.I)  #返回一个列表
```

随后我进行了改进，加入了textblob库，实现了名词的单复数还原，动词的词性还原和分词

```
def token_stream(line):

    li=TextBlob(line).words.singularize()
    li = ' '.join(li)  # 字符串
    terms = re.findall(r'\w+',li, re.I)
    result = []
    for word in terms:
        expected_str = Word(word)
        expected_str = expected_str.lemmatize("v")  # 将动词还原
        result.append(expected_str)
    return  result
```

2.开始匹配，mapper 即将term与相对应的text进行匹配，建立好索引 此时匹配的规则为 字典的 key 为 lineNum:term, value为 [词频]

```python
def mapper(lineNum, list):
    # dic key为 lineNum:term value为 词频
    dic = {}
    for item in list:
        key = ''.join([str(lineNum), ':', item])
        if key in dic.keys():
            ll = dic.get(key)
            ll.append(1)
            dic[key] = ll
        else:
            dic[key] = [1]

    return dic
```

3.开始结合词频，combiner 因为之前的出来的词频没有求和，现在是对每行词频进行求和，得到每行对应词频

```python
# 结合 词频
def combiner(dic):
    keys = dic.keys()
    tdic = {}
    for key in keys:
       # print(key)
        valuelist = dic.get(key) #得到记录 posting list
        count = 0
        for i in valuelist:
            count += i
        tdic[key] = count
    return tdic
```

4.开始将每个term对应的posting list进行合并，reducer，将之前的 字典 key为 lineNum:term, value为 [词频]，变为 key：term，value：[lineNum:词频]

```python
#将每个 term对应的 posting进行合并
def reducer(dic):
    keys = dic.keys()
    rdic = {}
    for key in keys:
        lineNum, kk = key.split(":")
        ss = ''.join([lineNum, ':', str(dic.get(key))]) #变成字符串
        if kk in rdic.keys():
            ll = rdic[kk]
            ll.append(ss)
            rdic[kk] = ll
        else:
            rdic[kk] = [ss]

    return rdic
```

5.进行排序，按term的首字母大小进行排序，从而建立起倒排索引

```python
#排序，返回一个列表
def shuffle(dic):
    dict = sorted(dic.items(), key=lambda x: x[0])
    return dict
```

结果图：

每个词对应了 它出现的文档（此时用行号作为文档）和在该文档下的词频

```
'satisfaction': ['19698:1'], 'satisfied': ['22142:1', '22442:1', '24102:1'], 'satisfy': ['1598:1', '1987:1'], 'satisfying': ['6406:1'],
'satk6NdTwW': ['20689:1'], 'satoonaaa': ['16674:1'], 'sauce': ['250:2', '3204:1', '3218:1', '4651:1', '5176:1', '5789:1', '6128:1',
'15350:1', '26763:1'], 'saudi': ['27442:1'], 'sauerkraut': ['29390:1'], 'saus': ['19545:1'], 'sausage': ['2938:1', '3209:1', '4855:1'],
'savage': ['6933:1'], 'savannah': ['22760:1'], 'save': ['1289:1', '2889:1', '2962:1', '3073:1', '3195:1', '3471:1', '3556:1', '4718:2',
'4908:1', '5203:1', '5514:1', '5763:1', '5836:1', '5859:1', '6029:1', '6094:1', '6220:1', '7091:1', '7470:1', '7471:1', '7486:1',
'7596:1', '11966:1', '12115:1', '12146:1', '12680:1', '12723:1', '13120:1', '14698:1', '16575:1', '17219:1', '17401:1', '17446:1',
'17817:1', '18403:1', '20685:1', '24553:1', '25319:1', '27000:1', '27953:1', '28418:1', '30098:1'], 'saveSFbay': ['28824:1'], 'saved':
['1426:1', '1610:1', '1887:1', '7109:1', '8174:1', '8222:1', '8245:1', '8282:1', '9058:1', '10324:1', '10325:1', '14252:1', '15808:1',
'20756:1', '25500:1', '25555:1', '25593:1', '29721:1', '30440:1'], 'saveolympicwrestling': ['11882:1', '11970:1', '11971:1', '12002:1',
'12054:1', '12115:1', '12120:1', '12127:1'], 'saveourforests': ['8094:1'], 'savers': ['10686:1'], 'saves': ['303:1', '2502:1', '9598:1',
'15716:1', '16720:1', '16728:1', '16764:1'], 'savewrestling': ['11981:1'], 'saving': ['186:1', '3073:1', '6258:1', '7878:1', '7900:1',
'8473:1', '12128:1', '13921:1', '15683:1', '15684:1', '16153:1', '17219:1', '21797:1'], 'savingelectricitybygoingtomaccas': ['6720:1'],
'savings': ['1755:1', '19513:1', '26991:1'], 'savingwater': ['6921:1'], 'savior': ['5056:1', '9244:1', '20692:1', '20715:1'],
'saviorhood': ['27233:1', '27235:1'], 'saviour': ['16386:1'], 'savvy': ['25937:1'], 'savvyreader': ['27717:1'], 'saw': ['494:1',
'1007:1', '1542:1', '1885:1', '1941:1', '2061:1', '2674:1', '2803:1', '3170:1', '3255:1', '3315:1', '3647:1', '3941:1', '4728:1',
'5875:1', '5978:1', '7011:1', '7651:1', '8546:2', '8695:1', '8818:1', '9621:1', '9638:1', '11046:1', '11382:1', '12009:1', '12572:1',
'14553:1', '14720:1', '14721:1', '15443:1', '15555:1', '15586:1', '16337:1', '16674:1', '17683:1', '18660:1', '19371:1', '19690:1',
```

用 tweetid 作为 文档id，并且把词频给去掉，实现标准输出

```
'wfhtvhwbra': ['623830347311128576'], 'wfinsurance': ['623668795282956292'], 'wfir': ['304200919585808384'], 'wfjuhdwr':
['299134611391401984'], 'wfla': ['308706940060569600'], 'wfld': ['326718321422581 76'], 'wfm': ['310248237699321856'], 'wfmy':
['308829812192075776'], 'wfnewscold': ['30838480187686913'], 'wfnewshinged': ['313244924919234 56'], 'wfnewsi': ['329517883279278 09'],
'wfpa9ayot0': ['312902569452257280'], 'wfpbkbafca': ['311263192154263552'], 'wfq33scprq': ['623979110860062720'], 'wfq9g2llpt':
['312092255064322048'], 'wfquy3f0ak': ['307664756146331648'], 'wft': ['29447012155920384', '29768355124617218', '30583328172154880',
'30724724736655361', '31442910696177665', '31656691342770176', '32921526181306368', '33333604473905153', '34241267537813504',
'307444202877370368'], 'wfx4znex5x': ['311472240468295681'], 'wfyxea4imp': ['316290161971453952'], 'wfz9ul7ojx': ['312597425435078656'],
'wg3oypuicw': ['315135000334958592'], 'wg5fsf4z': ['302422203335577600'], 'wg61hyeysa': ['305832738722807808'], 'wg7ceyzpsh':
['624157083571744768'], 'wgad3udxbg': ['314846398669524992'], 'wgaltv': ['309681494983573508'], 'wgaz': ['31450686512173056'],
'wgb83ovmpr': ['315105677934747648'], 'wgc': ['304800705024241666', '310503385621221377', '310541499270041601', '310573665379053568',
'310589754712420352', '310837235425611777', '310883888673202177', '310884069028294656', '310888200409333760', '310900640735977472',
'310953090494894080', '311008849584865281', '311061601342078976', '311163011207426048'], 'wgf': ['311473997885865984'], 'wgh7xole':
['299254723641495552'], 'wgj8rqimgd': ['312328201470889986'], 'wgme': ['303129199399542784'], 'wgmwuiqgg': ['306082979262906369'],
'wgntg5bm': ['297405337588342784'], 'wgowilo8su': ['307558518637080576'], 'wgp7kejrbl': ['626219141842751488', '626219502569570304',
'626221985576689664'], 'wguoq8cd': ['298894026118135808'], 'wgv01d5taj': ['310431478473424896'], 'wgwgi0ulgy': ['623251101320523776'],
'wgwsdh0lyg': ['309601325044359168'], 'wgxbviulye': ['309007721955487744'], 'wh': ['29610644344934400', '32051234642857984',
'33346093525762048', '297389151769219072', '297410559496826880', '297463470620807168', '299735852274298881', '304620417069494272',
'314730296144715776', '623477467928526848', '624781607044554752', '625458223806427136'], 'wh1sq0v1': ['298876170953428992'], 'wh2hwuo':
```

布尔查询

1.要求能支持 and，or，not操作，现在已经能得到倒排索引

2.先得到输入的字符串，变成列表进行遍历，判断是否存在 and，or，not的词语

3.将结果 answer_set变成集合，分别对 and or not 操作进行处理

4.and即对应集合的交集 即 intersection函数，or即对应 集合的并集 即 union函数，not即对应集合的差集difference函数，通过这三个函数来实现布尔查询

and 和 not 单独操作

```
Please input the word you want to search: House and may
{'1', '28886'}
Please input the word you want to search: House not may
{'26486', '15192', '10085', '8610', '7280', '2137', '2559', '25173', '26543', '30439', '5620', '1070', '2524', '14003', '4371', '5593',
 '14382', '25897', '26448', '8821', '26409', '30458', '2552', '292', '21798', '13019', '5971', '9039', '9995', '12723', '24587', '25395',
 '29794', '14966', '10011', '24733', '30444', '328', '22991', '9991', '14038', '28389', '8830', '15434', '9736', '8792', '2355', '2590',
 '4331', '9983', '18995', '15745', '2015', '16976', '28680', '10060', '24049', '25401', '23713', '25808', '8050', '8699', '5762', '9926',
 '13999', '2741', '9903', '8851', '2509', '15148', '9890', '25178', '2672', '2405', '14213', '9919', '27417', '9296', '27116', '17193',
 '18601', '9663', '15489', '30277', '8706', '10184', '8831', '25989', '29962', '14525', '21120', '8612', '780', '29219', '2937', '6181',
 '9964', '16220', '9655', '17019', '15438', '1971', '4641', '9965', '15320', '2167', '27021', '804', '28178', '19513', '29713', '29350',
 '28767', '9905', '4856', '17119', '9971', '25211', '8444', '16464', '3675', '53', '14285', '1778', '4370', '2652', '2664', '8003',
 '15479', '30469', '10709', '9938', '26369', '14983', '8585', '21811', '2726', '2592', '8807', '21886', '9674', '2545', '18078', '25988',
 '25676', '13699', '2788', '2691', '10097', '2839', '8739', '11316', '1317', '4883', '8589', '6214', '10066', '25469', '15485', '15782',
 '6069', '1840', '25741', '9111', '15876', '26911', '240', '2706', '15739', '2638', '16978', '26559', '15942', '9390', '18051', '1019',
 '15886', '15881', '8962', '2695', '8590', '8796', '10399'}
```

and 和 not 同时操作

```
Please input the word you want to search: House and may not the
set()
Please input the word you want to search: House and the not kill
{'10097', '9983', '18995', '11316', '15320', '4641', '16976', '23713', '6069', '19513', '1840', '17119', '9971', '25211', '9111', '4371',
 '14382', '4370', '25178', '2652', '2405', '2664', '28886', '26409', '10709', '9919', '2552', '26369', '9390', '18601', '5971', '9995',
 '24733', '2545', '22991', '9991', '9964', '9736', '10399', '6181', '17019', '2355'}
```

and or 和 not 同时操作

```
Please input the word you want to search: may or kill not House
{'25744', '4747', '27576', '3136', '24186', '14174', '28169', '11416', '18181', '8595', '4628', '17889', '19001', '25500', '28953',
 '12046', '23072', '8133', '22858', '21028', '24172', '27017', '6094', '14445', '22717', '25576', '25024', '9550', '11449', '29441',
 '3196', '15976', '7109', '1075', '28018', '13441', '16710', '20404', '28186', '11846', '12280', '7502', '12510', '27605', '30001',
 '13316', '7879', '87', '8874', '3354', '28798', '29951', '20030', '9869', '20360', '18242', '8832', '16663', '20350', '20888', '21618',
 '4916', '2354', '29243', '15139', '25800', '15298', '30187', '1309', '6260', '28015', '7516', '12355', '1120', '14502', '16001', '25850',
 '13487', '21003', '21936', '24584', '2708', '3989', '25259', '15935', '20533', '4159', '27210', '9144', '7417', '322', '20787', '9419',
 '20741', '22447', '14957', '15774', '5332', '13168', '27607', '22680', '24296', '12036', '12599', '14404', '14534', '27265', '21107',
 '330', '21343', '5110', '26651', '14184', '27274', '4482', '8948', '15519', '20889', '1307', '21932', '22692', '4390', '14265', '27550',
 '4886', '21911', '3288', '10682', '15032', '27262', '26121', '25555', '17279', '27409', '21900', '3055', '27421', '11920', '20009',
 '16486', '21775', '25593', '27234', '21947', '4272', '8094', '12356', '4623', '4998', '14911', '21692', '2218', '15780', '19129', '7880',
 '5569', '2057', '12157', '22141', '17186', '8811', '27092', '18056', '11584', '27304', '7886', '13750', '21937', '6281', '22277',
 '24410', '15900', '15908', '26167', '7078', '3200', '23755', '19339', '865', '8114', '4148', '16084', '22327', '4158', '27471', '23408',
 '20865', '4968', '18410', '15362', '4890', '13977', '9826', '12045', '13021', '19733', '2498', '15773', '28176', '7839', '20887',
 '25817', '9987', '22754', '5638', '24380', '25723', '27113', '21392', '15013', '9802', '13486', '15460', '12574', '12158', '19915',
 '27638', '16645', '15541', '16765', '18417', '1980', '17377', '9662', '4692', '4585', '21580', '16016', '9633', '14538', '9133', '27266',
 '22645', '28916', '4367', '27497', '2763', '7482', '14755', '19631', '5543', '27420', '27190', '9147', '11568', '11839', '17381',
 '30363', '11601', '18691', '26431', '22715', '11412', '26939', '2676', '8678', '14428', '13027', '28243', '4260', '28184', '203', '7901',
 '25450'}
Please input the word you want to search: may and House not kill
{'28886'}
```

改进

1. 增加了优先级操作，可以支持 A B C 三个单词之间的 and or not 的优先级操作，优先级关系为 not > and > or

```
Please input the word you want to search (all lower): House and may or friend
{'30369516349292544', '308665319952625664', '315307709203353600', '297465962049990657', '308007804097142785', '311809659650580480',
 '31721475786416129', '626048802789535744', '32530826499465216', '30656489479737344', '31263364470538240', '33269747457986560',
 '311923581149921280', '308436566827737088', '626416030861017088', '32886403880718337', '626058026047406080', '297167004656361472',
 '31244303820918784', '34682864171753472', '34985941571473408', '32793336188239874', '310173772038680576', '34982142656122881',
 '33581312786833409', '299054588261171200', '306811433419673600', '316828098257641472', '33820768919879680', '29247566075924480',
 '29740831803973632', '309128404693819392', '34629385424207872', '312692845838479360', '297559478239383552', '33973534418014209',
 '30185132027551744', '315194538463469568', '34723960016871424', '29428094217490432', '311156325499412482', '29790192529309696',
 '29657409127448576', '30095059768573953', '298447311758049282', '315146698240454656', '312297708885008384', '29047669389271042',
 '309740114567917568', '29936272629829632', '311233358074089472', '34102550642823168', '306526405271896064', '302839775625359360',
 '302107945087422465', '303678472876875777', '28965792812892160', '626464651220512768', '29551273996980224', '29637500142100480',
 '29725943555686400', '29961115022655489', '315181531956117504', '316302606484004864', '29817437197180928', '29549361918648320',
 '314388988830048256', '29909262767489024', '297469539778691072', '29990983902961664', '311002180633112576', '34596882256760832',
 '33292259323547649', '29577827254800385', '31234119782367233', '626060186126557189', '30355575934033920', '304735357772124161',
 '297693951849271298', '30958583155793920', '31700342240444418', '625887204615815169', '34381947379650560', '298247075676688385',
 '29883734643703808', '30545151508094976', '33241391375060992', '299904094204723200', '302190468043780096', '33468260049494016',
 '34892486522445824', '33516419014463488', '304813740954636288', '315148434694864896', '30071138327470081', '303582742061404160',
 '32877989695856641', '311535746450329600', '31387551839293441', '298166314374139905', '29928974628954114', '304920917979574273',
 '297485138407874560', '29655015152951297', '297566713438928896', '33226850880724992', '309960164537139200', '299130631001079808',
 '30947664312934400', '310045304688898048', '308986557518471169', '33551062635913216', '307605616497655809', '34763798942322688',
 '29283663715762178'}
```

2. 学习和实用了 textblob库，实用了 其中 words方法就进行分词，还有对名词的单复数处理，对动词进行词性还原，学习实用的方法图片如下图所示：

```
singularize() 变单数， pluralize () 变复数，用在对名词进行处理，且会考虑特殊名词单复数形式
```

```
>>> sentence = TextBlob('Use 4 spaces per indentation level.')
>>> sentence.words
WordList(['Use', '4', 'spaces', 'per', 'indentation', 'level'])
>>> sentence.words[2].singularize()
'space'
>>> sentence.words[-1].pluralize()
'levels'
```

Word 类： lemmatize() 方法 对单词进行词形还原，名词找单数，动词找原型。所以需要一次处理名词，一次处理动词

```
>>> from textblob import Word
>>> w = Word("octopi")
>>> w.lemmatize()      # 默认只处理名词
'octopus'
>>> w = Word("went")
>>> w.lemmatize("v")  # 对动词原型处理
'go'
```

3. 加入了统计词频（tf）和统计词的文档（df），便于实验2计算 文档和查询的分数

2. Ranked retrieval model

实验内容

1. 在Homework1.1的基础上实现最基本的Ranked retrieval model

1. • Input：a query (like Ron Weasley birthday)
2. • Output: Return the top K (e.g., K = 10) relevant tweets.
3. • Use SMART notation: lnc.ltc
4. • Document: logarithmic tf (l as first character), no idf and cosine normalization
5. • Query: logarithmic tf (l in leftmost column), idf (t in second column), nonormalization

2. 改进Inverted index

1. • 在Dictionary中存储每个term的DF
2. • 在posting list中存储term在每个doc中的TF with pairs (docID, tf)

3. 选做

• 支持所有的SMART Notations

实现思路

排序索引

一、 首先在实验一建立倒排索引的基础上，进行处理，建立了每一个term对应的 postings的结果，结果如实验一所示：

['306526841475325952'], 'tsc': ['624563218023845888'], 'tscent': ['302867005067894785', '304959186800881665', '308654490263879682',
'309113519117316096'], 'tsdbskqz': ['299855775801683968'], 'tsdeqzbhnz': ['623433742309322753'], 'tsdgcuok': ['302061438636265472'],
'tsfxiqly7b': ['304544504348819456'], 'tshirt': ['308680490771169281', '309074516280217600', '312249277223141376'], 'tsijbv3q':
['29603513705172992'], 'tsim': ['624881486014083072'], 'tsipra': ['623021324802347008', '623742690526494720', '623755021784518656',
'623769370527838208', '623845476174053376', '624043392792662016', '626069031892688896', '626325018666860544', '626325769455816704',
'626339501573582848'], 'tsmcj70mdi': ['625665787315929088'], 'tsn': ['301345303167123456'], 'tsn_sport': ['301345303167123456'],
'tspanu': ['29612016519876608'], 'tstoneee13': ['624710324885811200'], 'tsu': ['297741179682975745'], 'tsui': ['624881486014083072'],
'tsumaylim': ['318263699460747265'], 'tsunami': ['32574099448406018', '32595644770164736', '33394443545481216'], 'tsuolfud5i':
['307682250617659393'], 'tsxpnwov1p': ['624712887592992769', '624715039279304704', '624776460646117377'], 'tsxxe9wgmx':
['315186862903930880'], 'tsxy246ax4': ['623983363909496832'], 'tsyqwerd': ['301292765344591872'], 'tt': ['299254723641495552',
'307610414743707648', '307630274794119168', '308698383672016896', '315104243516334080', '32385356867309568', '623931136411062272'],
'tt0180093': ['33285922183327744'], 'tt1y6ialma': ['622872439559692288', '623137947420897280'], 'tt2gpy5vzi': ['624632050772013056'],
'tt2wwuoveq': ['625336165370040320'], 'ttapus': ['30743339112341504'], 'ttatham13': ['623580014450245632'], 'ttcsvehddb':
['308628842120114177'], 'ttd7vuzu': ['303521844990967808'], 'ttgc4o8uaz': ['626488655238926336'], 'ttgtalcv6d': ['307459268817326080'],
'ttgturauek': ['313352211415588865'], 'tthi': ['302199926190911488'], 'ttigqyxi': ['301773403193499649'], 'ttlaxzntdc':
['305077768159391745'], 'ttln9djq7k': ['307479607022612480'], 'ttmcfjup': ['301361358992179200'], 'ttnr6ff3fa': ['314150320370483201'],
'ttnz5159gj': ['310801806148116480'], 'ttot': ['625575630764011520', '626318752385253376'], 'ttqojpzd': ['303781187187535872'],
'ttqxy7qpjf': ['304352489149456384'], 'ttrwv5vtex': ['624728775595458560'], 'ttsznurzc6': ['305964527911051264'], 'ttu':

二、 在实验一的基础上，加上词频，即统计每一个文档内容的时候，记录在这一篇文档中词语的词频，mapper 即将term与相对应的text进行匹配，建立好索引 此时匹配的规则为 字典的 key为 lineNum:term, value为 [词频]，这样就得到了词频

```python
def mapper(lineNum, list):
    # dic key为 lineNum:term value为 词频
    dic = {}
    for item in list:
        key = ''.join([str(lineNum), ':', item])
        if key in dic.keys():
            ll = dic.get(key)
            ll.append(1)
            dic[key] = ll
        else:
            dic[key] = [1]

    return dic
```

三、 开始结合词频，combiner 因为之前的出来的词频没有求和，现在是对每行词频进行求和，得到每行对应词频，开始将每个term对应的posting list进行合并，reducer，将之前的 字典 key为 lineNum:term, value为 [词频]，变为 key：term，value：[lineNum:词频]，和实验一操作一样，然后再实现排序，按term的首字母大小进行排序，从而建立起倒排索引，其中每一个term对应了 tweetid和在这一篇 tweetid中出现的词频, 与实验一处理类似

```
# 结合 词频,得到每篇文章的词频
def combiner(dic):
    keys = dic.keys()
    tdic = {}
    for key in keys:
       # print(key)
        valuelist = dic.get(key) #得到记录 posting list
        count = 0
        for i in valuelist:
            count += i
        tdic[key] = count
    return tdic


#将每个 term对应的 posting进行合并
def reducer(dic):
    keys = dic.keys()
    rdic = {}
    for key in keys:
        lineNum, kk = key.split(":")
        ss = ''.join([lineNum, ':', str(dic.get(key))]) #变成字符串
        if kk in rdic.keys():
            ll = rdic[kk]
            ll.append(ss)
            rdic[kk] = ll
        else:
            rdic[kk] = [ss]
    for term in  rdic.keys():  # 对postings进行排序
        rdic[term].sort()
    return rdic

#排序，返回一个列表
def shuffle(dic):
    dict = sorted(dic.items(), key=lambda x: x[0])
    return dict
```

在posting list中存储term在每个doc中的TF with pairs (docID, tf),每个词对应了 它出现的文档序号和在该文档下的词频

结果如下图所示：

```
'301504921637908480:1'], 'vaisvcs8': ['302652151828709376:1'], 'vajtrozvdc': ['305062077297483776:1'], 'valcke': ['309658090758864896:1',
'624541835466158080:2', '624548831561162756:1', '624549871748562944:1', '624561884268744704:1', '624580901238734848:1',
'624581983365165057:1'], 'valde': ['309632174162837505:1'], 'valdez': ['623124039087861764:1'], 'valentine': ['301092407628423168:1',
'301355990262108160:1', '301854701413089281:1', '302014068166897664:1', '302021781487771649:1', '302066379551551488:1',
'302181899055542272:1', '302425407788032000:1', '347969247709839361:1'], 'valentinesday': ['302169987240579072:1'], 'valerie':
['29881782203580416:1', '626240855767142400:1'], 'valid': ['626521291097423872:1'], 'validate': ['310061859631947776:1'], 'validation':
['30909763394994176:1'], 'valija': ['308351711867715584:1'], 'valley': ['301504921637908480:1', '30707908433940480:1',
'307516814655643648:1', '309808951485161473:1', '313184640590561280:1', '32887217307258880:1', '625041326740877312:1',
'625912659867611136:1', '625950618327347200:1', '626210866464059392:1'], 'valuable': ['29747641034219520:1', '298451199852675072:1',
'298980969837391872:1', '314984114430304256:1'], 'valuation': ['304414733799342081:1'], 'value': ['29327931427786752:1',
'299230618988969985:1', '300992658682286080:1', '301363279962439680:1', '302127490531475456:1', '30222943480979456:1',
'302467447259267072:1', '304731037647372288:1', '306526841475325952:1', '307499441873645570:1', '30751177817722880:1',
'310012282937683969:1', '311214290780692481:1', '314621508473479168:1', '317384195842387968:1', '317442161119490048:1',
'317683362963787778:1', '317692829512134658:1', '317693752254808066:1', '317698252747186177:1', '317723330503405568:1',
'317772064125763584:1', '317815114470412288:1', '317831140901789697:1', '317870445728776192:1', '32104875970002944:1',
'32595979823751168:1', '32901908855263232:1', '33356246027345920:1', '33356335970000896:1', '33356503989616640:1', '33901964072853504:1',
'34374988265947136:1', '34455320864493568:1', '34456558750404608:1', '34588354041483264:1', '34766453005758464:1', '35045605193551872:1',
'623220189321007104:1', '623272408401575937:1', '623940019984543744:1', '624647188023390208:1', '624736610551083008:1'], 'valuerater':
['623625778517909506:1'], 'valve': ['30386385739317250:1', '30389268501241856:1', '30389512999796736:1'], 'vam': ['311435028594827264:1'],
```


四、统计词出现的总词频和文档频率，也就是对 term来计算 包含的 tweetid数目即 文档频率，每个tweetid中出现的词频求和，即总词频

```python
# 输入包含（词--id和词频），得到 词频和 文档频率
def tfAnddf(dic):
    pdic=defaultdict(dict)
    for word in dic.keys():
        lis=dic[word]
        tf=0
        df=0
        for u in lis:
            x,y=u.split(':')
            tf=tf+int(y)   #统计词频
            df=df+1        #统计文档频率
        pdic[word]=[str(tf),str(df)]
    #print(pdic)
    return pdic
```

五、对查询词 query，统计词频

```python
# 处理 query建立 词与词频的字典
def process_query(query):
    dic={}
    word=token_stream(query.lower())
    #print(word)
    for u in word:
        if u in dic.keys():
            dic[u]=dic[u]+1
        else:
            dic[u]=1
    return dic
```

六、 计算 inc.itc

使用如图所示的算法：

```
CosineScore(q)
float Scores[N] = 0
float Length[N]
for each query term t
do calculate w_{t,q} and fetch postings list for t
    for each pair(d, tf_{t,d}) in postings list
    do Scores[d] += w_{t,d} × w_{t,q}
Read the array Length
for each d
do Scores[d] = Scores[d]/Length[d]
return Top K components of Scores[]
```

首先循环query中出现的每一个词，计算出 词在query中的词频，词的逆文档频率，词在 tweetid中出现的词频，相乘，然后再对 词在tweetid出现的词频，求 l2范数(词向量在文档中的长度)，最后得出来 tf*idf 再除以 l2范数，得出最终的结果

```python
def do_RankSearch(query,doc,tdic):
    score={}
    length={}
    for term in query.keys():
        ll=query[term]
        ll=1+math.log(ll)  # query中的词频
        #print('ll: ',ll)
        if term in tdic.keys():  # 乘以 idf
           df=int(tdic[term][1])
         #  print('df: ',df)
           idf=math.log(30548/df)
           ll=ll*idf
         #  print('ll2: ',ll)
        if term in doc.keys():
            for postings in doc[term]:
                tweetid,tf=postings.split(':')
                tf=int(tf)
                tf = 1 + math.log(tf)
              # print('tf: ',tf)
                if tweetid  in score.keys():
                    score[tweetid]=score[tweetid]+ll*tf
                    length[tweetid]=length[tweetid]+tf**2
                else:
                    score[tweetid]=ll*tf
                    length[tweetid]=tf**2
    for tweetid in score.keys():
       score[tweetid]=score[tweetid]/math.sqrt(length[tweetid])


    return score
```

结果

以上是结果图，输入一个query，我会出现相应的分数

```
Please input the word you want to search (all lower):  you are a good boy
307424758105001984  :  6.995991433422537
625148071811256320  :  6.995991433422537
292721750305669l2  :  6.663492849878628
297764961386643457  :  6.6634928498786286
297817960632954881  :  6.6634928498786286
298252188550062081  :  6.6634928498786286
299228458914037760  :  6.6634928498786286
299883860873854976  :  6.6634928498786286
306189673976442880  :  6.6634928498786286
307396861784883200  :  6.6634928498786286
309700067361685505  :  6.6634928498786286
```

为了比较结果，我专门把相应的text也输出，可以看到里面有 query中的单词

```
Please input the word you want to search (all lower):  you are a good boy
Can't wait to live in a country where all my friends are treated equal - good on you UK for moving forward gay marriage
Hey, @HOTMessBarbie, you hear Taco Bell is being sued because their beef filling is only 35% beef?  I told you it makes a good enema!
Heads up to our Midwest fans, if you needed a snow blower now is a good time to pick one up.  Only 109 shipped to... http://t.co/PsbayFujJ1
@charliemcdrmott I'm doing this research into how real dreams actually are, and erm in my dreams you were a very good kisser...
How good is debt settlement over debt consolidation? Is it a more: Typically, you can do Debt Consolidation *BEFORE*... http://bit.ly/c0w8T2
Hu are you: (Scott) Breitbart has picked up the intriguing report "Chinese pianist plays propaganda tune a... http://bit.ly/gk0wbf #tcot
Japanese Kids Freak Out: This may just be a commercial for McDonald's, but you just can't fake this kind of raw emotion. http://su.pr/32PqNp
Why you say that ?? & #shoutout to everyone who wants to be her (lmao) RT @bbellz_702: Kim Kardashian is a real live slut..
McDonald's if you're reading this, good! Provide healthier menus so people have a chance to LIVE & NOT have a heart attack from your food!
RIP Barney. you were a great former first dog. @TheRealGDubya
Farewell NYC Mayor Ed Koch, as a kid you were the only mayor I met, sang Christmas carols with my class at Gracie Mansion, you held my hand
```

改进

1. 使用textblob库，对倒排索引建立，还有query查询都进行了处理，对名词的单复数处理，对动词进行词性还原，这样提高了效果，让结果更具有一般性

```python
def token_stream(line):
    #先变小写，然后名词变成单数
    line=line.lower()
    li=TextBlob(line).words.singularize()
    li = ' '.join(li)  # 列表变成 字符串
    terms = re.findall(r'\w+',li, re.I) # 只匹配字符和数字
    result = []
    for word in terms:
        expected_str = Word(word)
        expected_str = expected_str.lemmatize("v")  # 将动词还原
        result.append(expected_str)
    return  result
```

2. 加入了统计词频（tf）和 统计词的文档（df），求出 逆文档频率，并对结果进行了cosine 余弦函数归一化的处理，按照公式一步步得出，通过使用 ltc.lnc 模式，让结果更加正确
3. 在实验三中，我们使用 MAP，MRR，NDCG对实验二的查询结果进行了评价，具体结果请参照实验三，通过评价让查询的结果更加有说服力

3. Information Retrieval—Evaluation

实验内容

实现以下指标评价，并对HW1.2检索结果进行评价

1. • Mean Average Precision (MAP)
2. • Mean Reciprocal Rank (MRR)
3. • Normalized Discounted Cumulative Gain (NDCG)

实现思路——评价模型**

一、 对实验二查询结果进行评价，首先提取查询内容，在MB171-225.txt中有查询内容，查询内容为标签中的内容，如图所示：

```
<top>
<num> Number: MB171 </num>
<query> Ron Weasley birthday </query>
<querytime> Sat Mar 02 10:43:45 EST 2013 </querytime>
<querytweettime> 307878904759201794 </querytweettime>
<querydescription>
Find tweets regarding the birthday of fictional character Ron Weasley, Harry Potter's sidekick.
</querydescription>
</top>

<top>
<num> Number: MB172 </num>
<query> Merging of US Air and American </query>
<querytime> Sun Feb 17 16:14:40 EST 2013 </querytime>
<querytweettime> 303251140382973952 </querytweettime>
<querydescription>
Find information on the merger of US Airways and American airlines.
</querydescription>
</top>

<top>
<num> Number: MB173 </num>
<query> muscle pain from statins </query>
<querytime> Sat Mar 23 18:21:09 EDT 2013 </querytime>
<querytweettime> 315589058900418560 </querytweettime>
<querydescription>
Find mentions of muscle pain as a side effect of taking statin drugs.
```

我们将查询内容提取出来，并输实验二排序检索模型中,得到 查询出来的 前K个 相关的 文档，写入final_result.txt，然后进行评价

```
    with open('F:\\信息检索\\evaluation\\final_result.txt', 'w', encoding='utf-8') as f_out:
        with open('F:\\信息检索\\evaluation\\MB171-225.txt', 'r', encoding='utf-8') as file:
            lis = []
            for line in file.readlines():
                if line.find('<query>') != -1:
                    line = re.sub('<query>|</query>', '', line)
                    line = line.strip()
                    lis.append(line)
        id=171
        for query in lis:
```

```
            query = process_query(query)
            score=do_RankSearch(query,dic,tf_dic)
            score = dict(sorted(score.items(), key=lambda x: x[1],
reverse=True))
            for i, key in enumerate(score.keys()):
                text=str(id)+' '+key
                f_out.write(text+'\n')
                if i ==k:
                    break
            id=id+1
```

二、 在这里，我顺便提一下 几个文件 其中 qrels.txt 即标准答案的输出结果文件，result.txt 即 将标准输出结果进行提取，然后再进行评价，可以得到 RightAnwerEvaluation.txt文件 ，即标准答案的结果，如下图所示：

MAP = 0.6148422817122279
MRR = 0.07820415596074004
NDCG = 0.756819929645465

三、编写评价函数，MAP,MRR,NDCG

MAP，先得到qrels.txt 即标准答案的输出结果的 tweetid(docid)，再得到你的结果的 tweetid(docid),使用下图的公式，浏览每一项文档，分子为相关文档的个数，分母为浏览文档的个数，此时求出每一项的 AP，最后再进行求均值 得到 MAP

$$\mathrm{AP}(q_j) = \frac{1}{m_j} \sum_{k=1}^{m_j} Precision(R_{jk})$$

If a relevant doc is not retrieved at all, the Precision(...) is considered 0

Mean average precision (MAP) averages over multiple queries

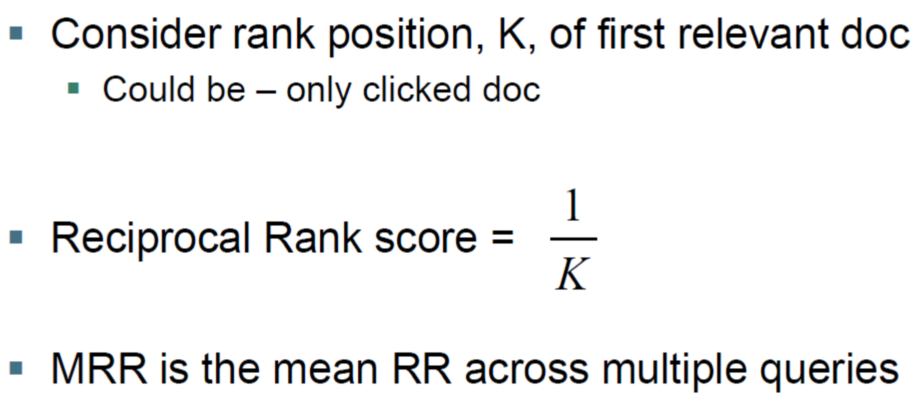
$$\mathrm{MAP}(Q) = \frac{1}{|Q|} \sum_{j=1}^{|Q|} \mathrm{AP}(q_j)$$

代码如下：

```
def MAP_eval(qrels_dict, test_dict, k = 100):
    AP_result = []
    file.write('MAP_evaluation: ' + '\n')
    for query in qrels_dict:
        test_result = test_dict[query] # 得到[docid,docid,docid,.......]
        true_list = set(qrels_dict[query].keys())  # 得到
[docid,docid,docid,.......]
        #print(len(true_list))
        #length_use = min(k, len(test_result), len(true_list))
        length_use = min(k, len(test_result))
        if length_use <= 0:
            print('query ', query, ' not found test list')
            return []
        P_result = []
```

```
        i = 0
        i_retrieval_true = 0
        for doc_id in test_result[0: length_use]:
            i += 1
            if doc_id in true_list:
                i_retrieval_true += 1
                P_result.append(i_retrieval_true / i)
                #print(i_retrieval_true / i)
        if P_result:
            AP = np.sum(P_result) / len(true_list)
            print('query:', query, ',AP:', AP)
            file.write('query' + str(query) + ', AP: ' + str(AP)+'\n')
            AP_result.append(AP)
        else:
            print('query:', query, ' not found a true value')
            AP_result.append(0)
    MAP=np.mean(AP_result)
    file.write('MAP' + ' = ' + str(MAP) + '\n')
    return MAP
```

MRR，和MAP 步骤类似，使用倒数的方法，先得到qrels.txt 即标准答案的输出结果的tweetid(docid)，再得到你的结果的 tweetid(docid),使用下图的公式，分子为一，分母为相关文档的位置，得到RR，最后再对所有RR 求和

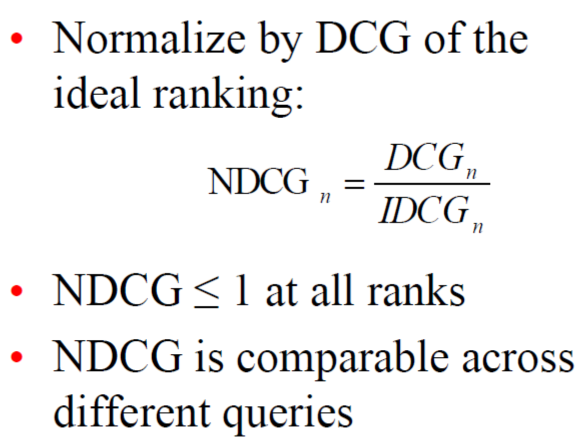


代码如下：

```
def MRR_eval(qrels_dict, test_dict, k = 100):
    RR_result = []
    file.write('\n')
    file.write('\n')
    file.write('MRR_evaluation: ' + '\n')
    for query in qrels_dict:
        test_result = test_dict[query] # 得到[docid,docid,docid,.......]
        true_list = set(qrels_dict[query].keys())  # 得到
[docid,docid,docid,.......]
        length_use = min(k, len(test_result))
        if length_use <= 0:
            print('query ', query, ' not found test list')
            return []
        P_result = []
        i_retrieval_true = 0
        for doc_id in test_result[0: length_use]:
```

```
            if doc_id in true_list:
                i_retrieval_true += 1
                P_result.append(1/ i_retrieval_true)
        if P_result:
            RR = np.sum(P_result) / len(true_list)
            print('query:', query, ',RR:', RR)
            file.write('query' + str(query) + ', RR: ' + str(RR)+'\n')
            RR_result.append(RR)
        else:
            print('query:', query, ' not found a true value')
            RR_result.append(0)
    MRR=np.mean(RR_result)
    file.write('MRR' + ' = ' + str(MRR) + '\n')
    return MRR
```

NDCG: 先算 DCG 为累计的相关性之和,再除以 位置的以2为底的对数，算 IDCG 为 将从大到小排序之后的 DCG，然后再用下图 公式 求出 NDCG



代码如下:

```
def NDCG_eval(qrels_dict, test_dict, k = 100):
    NDCG_result = []
    file.write('\n')
    file.write('\n')
    file.write('NDCG_evaluation: '+'\n')
    for query in qrels_dict:
        test_result = test_dict[query] # 得到[docid,docid,docid,.......]
        # calculate DCG just need to know the gains of groundtruth
        # that is [2,2,2,1,1,1]
        true_list = list(qrels_dict[query].values())
        true_list = sorted(true_list, reverse=True)
        i = 1
        DCG = 0.0
        IDCG = 0.0
        # maybe k is bigger than arr length
        length_use = min(k, len(test_result), len(true_list))
        if length_use <= 0:
            print('query ', query, ' not found test list')
            return []
        for doc_id in test_result[0: length_use]:
            i += 1
```

```
            rel = qrels_dict[query].get(doc_id, 0)
            DCG += (pow(2, rel) - 1) / math.log(i, 2)
            IDCG += (pow(2, true_list[i - 2]) - 1) / math.log(i, 2)
        NDCG = DCG / IDCG
        print('query', query, ', NDCG: ', NDCG)
        file.write('query'+str(query)+', NDCG: '+str(NDCG)+'\n')
        NDCG_result.append(NDCG)
    NDCG=np.mean(NDCG_result)
    file.write('NDCG' + ' = ' + str(NDCG) + '\n')
    return NDCG
```

四、输入我的结果final_result.txt 为171到225 所有查询词的 相关文档，我每个查询词取了100个，使用 上方的三种评价方式进行评价，MAP，MRR，NDCG得到结果

结果

每一个 query 中的 MAP 如下图，其余 MRR,NDCG 就不一一展示了

```
MAP_evaluation:
query171, AP: 0.981026192540312
query172, AP: 0.3412969283276451
query173, AP: 0.40917081589163173
query174, AP: 0.9166666666666666
query175, AP: 0.355221446036527
query176, AP: 0.9303678242429785
query177, AP: 0.8771929824561403
query178, AP: 0.4189667343368487
query179, AP: 0.9955151021717751
query180, AP: 0.153713298791019
query181, AP: 0.9346178286129265
query182, AP: 0.19305019305019305
query183, AP: 0.425531914893617
query184, AP: 0.5581307888233937
query185, AP: 0.8571428571428571
query186, AP: 0.7623933790628036
query187, AP: 1.0
query188, AP: 0.6223402856019554
query189, AP: 0.6877121999282026
query190, AP: 0.7484536854120513
query191, AP: 0.9134163651614368
query192, AP: 0.6581596067819252
query193, AP: 0.2615965863128985
query194, AP: 1.0
query195, AP: 0.2094923541784976
query196, AP: 0.6991656915685222
```

最终求和之后：

MAP = 0.5872561884769507
MRR = 0.07616857127966696
NDCG = 0.7555221939539475

结果比 标准答案小了一些，但基本上接近，实现的查询是有效的

改进

1. 使用textblob库，对 query查询都进行了处理，对名词的单复数处理，对动词进行词性还原，这样提高了效果，让结果更具有一般性

```
def token_stream(line):
    #先变小写，然后名词变成单数
    line=line.lower()
    li=TextBlob(line).words.singularize()
    li = ' '.join(li)  # 列表变成 字符串
    terms = re.findall(r'\w+',li, re.I) # 只匹配字符和数字
    result = []
    for word in terms:
        expected_str = Word(word)
        expected_str = expected_str.lemmatize("v")  # 将动词还原
        result.append(expected_str)
    return  result
```

2. 加入了统计词频（tf）和 统计词的文档（df），求出 逆文档频率，并对结果进行了cosine 余弦函数归一化的处理，通过使用 ltc.lnc 模式，大幅度提高了结果，比使用普通的对 按照出现的query中词个数之和排序的效果要好很多

4. Information Retrieval-Clustering with sklearn

实验内容(First)

1. 读入数据集：
   -sklearn.datasets.load_digits
   -sklearn.datasets.fetch_20newsgroups

Load and return the digits dataset (classification).

Each datapoint is a 8x8 image of a digit.

Classes	10
Samples per class	~180
Samples total	1797
Dimensionality	64
Features	integers 0-16

Load the filenames and data from the 20 newsgroups dataset (classification).

Download it if necessary.

Classes	20
Samples total	18846
Dimensionality	1
Features	text

2. 学习经典聚类方法，网址为：

   https://scikit-learn.org/stable/modules/clustering.html#

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

3. 使用经典的聚类方法来对两个数据集进行聚类，使用下面的评价方法进行评价

- Evaluation
  - labels_true and labels_pred
    - >>> from sklearn import metrics
    - >>> labels_true = [0, 0, 0, 1, 1, 1]
    - >>> labels_pred = [0, 0, 1, 1, 2, 2]
  - Normalized Mutual Information (NMI)
    - >>> metrics.normalized_mutual_info_score(labels_true, labels_pred)
  - Homogeneity: each cluster contains only members of a single class
    - >>> metrics.homogeneity_score(labels_true, labels_pred)
  - Completeness: all members of a given class are assigned to the same cluster
    - >>> metrics.completeness_score(labels_true, labels_pred)

实验思路

数据处理

一、对 sklearn.datasets.load_digits 手写识别数据集进行处理
得到 一个 1797*64维度的矩阵，一共1797个64像素的图片，然后需要对矩阵进行标准化处理，使用scale函数，即：(X- 均值) / 标准差，数据一共十个类别，分别是数字 0-9， 一共十个数字，然后进行处理，准备聚成10类，评价函数为下图：

```
def bench_k_means(estimator, name, data):
    t0 = time()
    estimator.fit(data)
    print('%-9s\t%.2fs\t%i\t%.3f\t%.3f\t%.3f\t%.3f\t%.3f\t%.3f'
          % (name, (time() - t0), estimator.inertia_,
             metrics.homogeneity_score(labels, estimator.labels_),
             metrics.completeness_score(labels, estimator.labels_),
             metrics.v_measure_score(labels, estimator.labels_),
             metrics.adjusted_rand_score(labels, estimator.labels_),
             metrics.adjusted_mutual_info_score(labels,  estimator.labels_,
                                                average_method='arithmetic'),
             metrics.silhouette_score(data, estimator.labels_,
                                      metric='euclidean',
                                      sample_size=sample_size)))
```

聚类函数

二、 使用 聚类函数：

1. K-Means 聚类

   bench_k_means(KMeans(init='k-means++', n_clusters=n_digits, n_init=10),name="k-means++", data=data)
2. 相似性传播

af = AffinityPropagation().fit(data)
bench_show(AffinityPropagation(),name="AffinityPropagation", data=data)
3. 均值漂移

   bandwidth = estimate_bandwidth(data,quantile=0.2,n_samples=500)
   bench_show(MeanShift(bandwidth=bandwidth,bin_seeding=True),
   name="MeanShift", data=data)
4. 光谱聚类

   使用PCA数据降维

   pca=PCA(n_components=n_digits).fit_transform(data)
   bench_show3(SpectralClustering(n_digits),name="SpectralClustering", data=pca)
5. 分层聚类

   ward = AgglomerativeClustering(n_clusters=n_digits, linkage='ward')
   ward.fit(data)

   bench_show(AgglomerativeClustering(n_clusters=n_digits, linkage='ward'),
   name="AgglomerativeClustering", data=data)
6. 基于密度的聚类

   db = DBSCAN().fit(data)

   bench_show3(DBSCAN(),name="DBSCAN", data=data)

7. 光学聚类

   clust = OPTICS(min_samples=50, xi=.05, min_cluster_size=.05)

   bench_show(OPTICS(min_samples=50, xi=.05, min_cluster_size=.05),name="OPTICS", data=data)
8. 高斯混合模型

   gmm = mixture.GaussianMixture(n_components=n_digits, covariance_type='full').fit(data)

   bench_show2(mixture.GaussianMixture(n_components=n_digits, covariance_type='full'),name="Gaussian", data=data)
9. Birch

   brc = Birch(branching_factor=50, n_clusters=n_digits, threshold=0.5, compute_labels=True)

   brc.fit(data)

   bench_show2(Birch(branching_factor=50, n_clusters=n_digits, threshold=0.5, compute_labels=True),name="Birch", data=data)

结果

```
init          time      inertia homo      compl     v-meas    ARI  AMI  silhouette
k-means++     0.21s     69406     0.603     0.652     0.627     0.466     0.623     0.165
AffinityPropagation 4.29s     0.932     0.460     0.616     0.154     0.573     0.058
MeanShift     0.15s     0.009     0.257     0.017     0.000     0.006     0.556
SpectralClustering   4.70s    0.001     0.271     0.001     -0.000    -0.000
AgglomerativeClustering 0.17s    0.758     0.836     0.796     0.664     0.793     0.112
DBSCAN        0.41s     0.000     1.000     0.000     0.000     -0.000
OPTICS        3.97s     0.134     0.967     0.235     0.045     0.233     0.036
Gaussian      0.53s     0.609     0.699     0.651     0.434     0.647     0.115
Birch         0.28s     0.758     0.836     0.796     0.664     0.793     0.124
```

实验内容(second)

数据处理

对 sklearn.datasets.fetch_20newsgroups 进行处理，一共有20类新闻数据

alt.atheism	2019/11/10 12:32	文件夹
comp.graphics	2019/11/10 12:32	文件夹
comp.os.ms-windows.misc	2019/11/10 12:32	文件夹
comp.sys.ibm.pc.hardware	2019/11/10 12:32	文件夹
comp.sys.mac.hardware	2019/11/10 12:33	文件夹
comp.windows.x	2019/11/10 12:33	文件夹
misc.forsale	2019/11/10 12:33	文件夹
rec.autos	2019/11/10 12:33	文件夹
rec.motorcycles	2019/11/10 12:33	文件夹
rec.sport.baseball	2019/11/10 12:33	文件夹
rec.sport.hockey	2019/11/10 12:33	文件夹
sci.crypt	2019/11/10 12:33	文件夹
sci.electronics	2019/11/10 12:33	文件夹
sci.med	2019/11/10 12:33	文件夹
sci.space	2019/11/10 12:33	文件夹
soc.religion.christian	2019/11/10 12:33	文件夹
talk.politics.guns	2019/11/10 12:34	文件夹
talk.politics.mideast	2019/11/10 12:34	文件夹
talk.politics.misc	2019/11/10 12:34	文件夹
talk.religion.misc	2019/11/10 12:34	文件夹

categories = ['alt.atheism', 'talk.religion.misc','comp.graphics', 'sci.space'],从20个类别中，提取4个类别的数据（包括训练集和测试集），进行处理，我们可以选择 HashingVectorizer 和 TfidfVectorizer 两种方式对文本进行 向量化，这里我选择使用 TfidfVectorizer，这样就可以得到了 X矩阵 n_samples: 3387, n_features: 10000

实验思路

五、使用 降维函数，进行降维 TruncatedSVD

```python
def decompostion(X):
    svd = TruncatedSVD(n_digits)
    normalizer = Normalizer(copy=False)
    lsa = make_pipeline(svd, normalizer)

    X = lsa.fit_transform(X)

    # print("done in %fs" % (time() - t0))
    #
    # explained_variance = svd.explained_variance_ratio_.sum()
    # print("Explained variance of the SVD step: {}%".format(
    #     int(explained_variance * 100)))
    return X
```

使用 评价函数

```
def bench_k_means(estimator, name, data):
    t0 = time()
    estimator.fit(data)
    print('%-9s\t%.2fs\t%i\t%.3f\t%.3f\t%.3f\t%.3f\t%.3f'
          % (name, (time() - t0), estimator.inertia_,
             metrics.homogeneity_score(labels, estimator.labels_),
             metrics.completeness_score(labels, estimator.labels_),
             metrics.v_measure_score(labels, estimator.labels_),
             metrics.adjusted_rand_score(labels, estimator.labels_),
             metrics.silhouette_score(data, estimator.labels_,
                                      metric='euclidean',
                                      sample_size=10000)))
```

聚类函数

六、使用聚类函数：

1. K-Means 聚类

   km=MiniBatchKMeans(n_clusters=true_k, init='k-means++', n_init=1,
                    init_size=1000, batch_size=1000, verbose=opts.verbose)
   bench_k_means(km,name="k-means++", data=X)
2. 相似性传播

   数据没有降维

   bench_show(AffinityPropagation(),name="AffinityPropagation",data=X)
3. 均值漂移

   bandwidth = estimate_bandwidth(decompostion(X), quantile=0.2, n_samples=500)

   bench_show(MeanShift(bandwidth=bandwidth,bin_seeding=True),
          name="MeanShift", data=decompostion(X))
4. 光谱聚类

   bench_show3(SpectralClustering(n_digits),
          name="MeanShift", data=decompostion(X))
5. 分层聚类

   bench_show(AgglomerativeClustering(n_clusters=n_digits,
   linkage='ward'),name="AgglomerativeClustering", data=decompostion(X))
6. 基于密度的聚类

   bench_show3(DBSCAN(),name="DBSCAN",data=X)

7. 光学聚类

   bench_show(OPTICS(),name="OPTICS", data=decompostion(X))
8. 高斯混合模型

   bench_show2(mixture.GaussianMixture(n_components=n_digits,
   covariance_type='full'),name="Gaussian", data=decompostion(X))

9. Birch

bench_show2(Birch(branching_factor=50, n_clusters=n_digits, threshold=0.5, compute_labels=True),name="Birch", data=decompostion(X))

七、 K-means 输出质心函数,输出聚类中心的文本

```
if not opts.use_hashing:
    print("Top terms per cluster:")

    if opts.n_components:
        original_space_centroids = svd.inverse_transform(km.cluster_centers_)
        order_centroids = original_space_centroids.argsort()[:, ::-1]
    else:
        order_centroids = km.cluster_centers_.argsort()[:, ::-1]

    terms = vectorizer.get_feature_names()
    for i in range(true_k):
        print("Cluster %d:" % i, end='')
        for ind in order_centroids[i, :10]:
            print(' %s' % terms[ind], end='')
        print()
```

结果

聚类结果:

```
init          time    inertia homo    compl   v-meas  ARI AMI silhouette
k-means++     0.08s   3275    0.473   0.480   0.476   0.380   0.007
AffinityPropagation 13.14s  0.885   0.191   0.314   0.008   0.079
MeanShift     0.32s   0.585   0.624   0.604   0.614   0.477
MeanShift     1.61s   0.526   0.585   0.554   0.507
AgglomerativeClustering 0.43s   0.576   0.653   0.612   0.603   0.434
DBSCAN        0.54s   0.002   0.168   0.003   -0.000
OPTICS        3.01s   0.230   0.167   0.193   0.009   -0.494
Gaussian      0.03s   0.584   0.605   0.594   0.611   0.485
Birch         0.04s   0.561   0.590   0.575   0.563   0.465
```

K-means的 质心文章:

```
Top terms per cluster:
Cluster 0: god sandvik com people jesus don morality christian kent say
Cluster 1: space access nasa henry digex pat toronto alaska gov shuttle
Cluster 2: graphics image thanks university 3d files images file program gif
Cluster 3: com article posting nntp sgi host university god livesey keith
```