Distributed Data Analytics
Exercise 2
Krithika Murugesan - 277537

Exercise 1 : Data cleaning and text tokenization

The given dataset is documents categorized into twenty news groups, Each category has several text files, these files are read into data frame with long with the labels. The root process(rank = 0) takes care of this, then it splits the data into more or less equal chunks and sends it to the other workers. These workers clean the text in the chunks of data given to them and they are sent back back to the root process. The NLP methods for the data cleansing is as follows:

```
def textCleanse(samples):
    #Split into words
    tokens = word_tokenize(str(samples))
    # convert to lower case
    tokens = [w.lower() for w in tokens]
    # remove all tokens that are not alphabetic
    words = [word for word in tokens if word.isalpha()]
    #Filter stop words
    stop_words = set(stopwords.words('english'))
    words = [w for w in words if not w in stop_words]
    return words
```

Though the workers are called in a loop, the barrier function is used to synchronize the whole process,

```
#print(np.array_split(folder,size))
comm.barrier()

#Root process distributes data to workers and gets the output
if rank == 0:
    rawData = readData()
    text = np.array(rawData['Text'])[0:1]
    print("Data read",text)
    #Splitting the data to remaining workers
    split = np.array_split(text,size-1)
    #For each worker; assign chunk of data
    for i in range(1,size):
        t1 = MPI.Wtime()
        comm.send(split[i-1],dest = i)
        #Receive the processed data
        tokenData.extend(comm.recv(source= i))
        t2 = MPI.Wtime()
        time.append(t2-t1)
    print("Time taken is ",np.mean(time))
    print("After processing ",tokenData)
    #Writing the data to save time
    #with open("cleanedData.csv", 'w') as f:
        #writer = csv.writer(f)
        #writer.writerows(tokenData)

else:
    #Workers
    partFile = comm.recv(source = 0)
    comm.send(list(map(textCleanse,partFile)),dest = 0)

comm.barrier()
```

This process is time and resource consuming, so the tokenized output is written to a text file, from where it can be read easily to execute the upcoming exercises feasibly enough number of times. The time required with different number of workers is as follows.

Processes	Time
2	115.06
3	71.11
4	61.95

The input to the function is

[list([b'Xref: cantaloupe.srv.cs.cmu.edu alt.news-media:739 alt.politics.elections:6015 talk.politics.misc:124146\n', b'Path: cantaloupe.srv.cs.cmu.edu!rochester!rutgers!usc!nic.csu.net!vmsb.is.csupomona.edu!mpye\n', b'From: mpye@vmsb.is.csupomona.edu\n', b'Newsgroups: alt.news-media,alt.politics.elections,talk.politics.misc\n', b'Subject: Re: Media horrified at Perot investigating Bush!\n', b'Message-ID: <1992Jun25.230454.1@vmsb.is.csupomona.edu>\n', b'Date: 26 Jun 92 07:04:54 GMT\n', b'References: <TOM.92Jun23082822@amber.ssd.csd.harris.com> \n', b' <visser.709454466@convex.convex.com>
<1992Jun25.151502.1@vmsb.is.csupomona.edu>
<visser.709530365@convex.convex.com>\n', b'Organization: California State Polytechnic University, Pomona\n', b'Lines: 29\n', b'Nntp-Posting-Host: acvax1\n', b'Nntp-Posting-User: cvads008\n', b'\n', b'visser@convex.com (Lance Visser) writes:\n', b"> +>I can't find my source.\n", b"> +>But. If you state that you will retract your claim, I'll go dig one up\n", b'> +>at the library. Fair enough?\n', b'> \n', b"> \tARE YOU SERIOUS? I'm not talking about retracting anything until\n", b'> you have produced SOMETHING.\n', b'> \n', b'> \tIf you were not just talking off the top of your head, I would\n', b'> assume that you have SOME memory of what your source is.\n', b'> \n', b'> \tPUT UP NOW without conditions!\n', b'\n', b'\n', b'Yes, very serious. I claim that I can substantiate my statement that\n', b"Rudman says he doesn't believe Perot was investigating him. You claim\n", b'Perot was investigating him. If you will state that you were in error\n', b"on this point, provided I produce the source, I'll go dig it up.\n", b'\n', b"Now give me one reason why I should go to the trouble if you won't\n", b"agree to this? It is simple enough you know. But I don't have time\n", b"to waste if you'll just blow it off with more of the tripe you usually\n", b'post.\n', b'\n', b'\n', b'\n', b'---\n', b'Michael Pye\n', b'email: mpye@csupomona.edu\n'])]

The output is

[['rochester', 'rutgers', 'usc', 'mpye', 'media', 'horrified', 'perot', 'investigating', 'bush', 'jun', 'b', 'california', 'state', 'polytechnic', 'university', 'lance', 'visser', 'writes', 'b', 'ca', 'find', 'b', 'state', 'retract', 'claim', 'go', 'dig', 'one', 'b', 'library', 'fair', 'enough', 'b', 'b', 'serious', 'talking', 'retracting', 'anything', 'b', 'produced', 'b', 'b', 'talking', 'top', 'head', 'b', 'assume', 'memory', 'source', 'b', 'b', 'without', 'conditions', 'serious', 'claim', 'substantiate', 'statement', 'b', 'rudman', 'says', 'believe', 'perot', 'investigating', 'investigating', 'state', 'b', 'point', 'provided', 'produce', 'source', 'go', 'dig', 'b', 'give', 'one', 'reason', 'go', 'trouble', 'b', 'agree', 'simple', 'enough', 'know', 'b', 'waste', 'blow', 'tripe', 'b', 'mpye']]

Exercise 2 : Calculate Term Frequency

The term frequency of a token is the number of times the token appears in the document normalized by the total number of tokens in the document. Since, each document is not independent on others for this computation, it can be easily parallelized, The worker reads the cleaned the data, divides it into chunks of similar size and distributes to the workers. Each worker has a set of documents for which they compute the TF and send back the results to the root, where all the results are concatenated. The Term Frequency is calculated by the following code snippet,

```
#calculate Term Frequency
def tfidCalc(vector):
    tfList = []
    for every in vector:
        #For each document, counting the occurence of words
        partial = (Counter(str(every).split(',')))
        #Total number of words in the doc
        total = len((str(every).split(',')))
        #Saving the Term frequency to tfList
        tfList.append({k: v / total for k, v in partial.items()})
    return(tfList)
```

The function is wrapped in a map from where it is being called, so it will iterate over the all rows, partial is the frequency of each token in a document and total is the total number of tokens in the document. From which Term frequency can be easily computed. The MPI framework is

```
if rank == 0:
    #Read tokenized data
    text = np.array((pd.read_csv('cleanedData1.csv',header =None,sep='delimiter',engine = 'python')).values)
    text =(text)
    print("Data read",len(text))
    #Split data
    split = np.array_split(text,size-1)
    #Send data to workers
    for i in range(1,size):
        t1 = MPI.Wtime()
        comm.send(split[i-1],dest = i)
        tfId.extend(comm.recv(source= i))
        t2 = MPI.Wtime()
        time.append(t2-t1)
    print("Time taken is ",np.mean(time))
    #print("tfID",*tfId, sep = ",")

else:
    #worker
    partFile = comm.recv(source = 0)
    comm.send(list(map(tfidCalc,partFile)),dest = 0)
```

Varying the number of workers still returns the same term frequencies, with variance in the time required for computation. Considering one of the tokens, '**rochester**' has the Tf of 0.01124 irrespective of the number of workers used. The full document is shown in the ipython notebook.

Processes	Time
2	2.48154
3	1.63801
4	1.22058

The sample output is

tfID,[{'rochester': 0.011235955056179775, 'rutgers': 0.011235955056179775, 'usc': 0.011235955056179775, 'mpye': 0.02247191011235955, 'media': 0.011235955056179775, 'horrified': 0.011235955056179775, 'perot': 0.02247191011235955, 'investigating': 0.033707865168539325, 'bush': 0.011235955056179775, 'jun': 0.011235955056179775, 'b': 0.20224719101123595, 'california': 0.011235955056179775, 'state': 0.033707865168539325, 'polytechnic': 0.011235955056179775, 'university': 0.011235955056179775, 'lance': 0.011235955056179775, 'visser': 0.011235955056179775, 'writes': 0.011235955056179775, 'ca': 0.011235955056179775, 'find': 0.011235955056179775, 'retract': 0.011235955056179775, 'claim': 0.02247191011235955, 'go': 0.033707865168539325, 'dig': 0.02247191011235955, 'one': 0.02247191011235955, 'library': 0.011235955056179775, 'fair': 0.011235955056179775, 'enough': 0.02247191011235955, 'serious': 0.02247191011235955, 'talking': 0.02247191011235955, 'retracting': 0.011235955056179775, 'anything': 0.011235955056179775, 'produced': 0.011235955056179775, 'top': 0.011235955056179775, 'head': 0.011235955056179775, 'assume': 0.011235955056179775, 'memory': 0.011235955056179775, 'source': 0.02247191011235955, 'without': 0.011235955056179775, 'conditions': 0.011235955056179775, 'substantiate': 0.011235955056179775, 'statement': 0.011235955056179775, 'rudman': 0.011235955056179775, 'says': 0.011235955056179775, 'believe': 0.011235955056179775, 'point': 0.011235955056179775, 'provided': 0.011235955056179775, 'produce': 0.011235955056179775, 'give': 0.011235955056179775, 'reason': 0.011235955056179775, 'trouble': 0.011235955056179775, 'agree': 0.011235955056179775, 'simple': 0.011235955056179775, 'know': 0.011235955056179775, 'waste': 0.011235955056179775, 'blow': 0.011235955056179775, 'tripe': 0.011235955056179775}]

Exercise 3 : Inverse Document Frequency

The inverse document frequency is the log of number of documents in the corpus divided by the frequency of a token across all documents, this is relative to the whole corpus, hence parallelization is not as easy as in the previous case. First, data is processed to get the unique words in a document, and this list is concatenated and counter is applied to the new list, which gives the frequency of each token across the different documents. This dictionary is sent to all workers from the root process. The documents are also evenly distributed across the workers, so they can calculate the inverse document frequency for all the tokens they have. The data processing and IDF code snippets are

```
def dataProcess(text):
    keyFin = []
    for each in text:
        for every in each:
            keys = []
            partial = (Counter(str(every).split(',')))
            for key,value in partial.items():
                keys.append(key)
            keyFin.append(keys)
    docs = sum(keyFin,[])#One dict
    Freq = dict(Counter(docs))#Total dict
    return(keyFin,Freq)

def idfCalc(keyFin,Freq,nr):
    idfList = []
    for each in keyFin:
        new = Counter(each)
        #Dividing the total by the value from the document with occurence of words in corpus
        idfList.append({k: np.log(v*(nr/Freq[k])) for k, v in new.items()})
    return(idfList)
```

The root and workers are synchronized as follows

```
comm.Barrier()

if rank == 0:
    #Sending data to workers
    text = np.array((pd.read_csv('cleanedData1.csv',header =None,sep='delimiter',engine = 'python')).values)
    rows = len(text)
    print("Data read",rows)
    finData,totFreq = dataProcess(text)
    split = np.array_split(finData,size-1)
    for i in range(1,size):
        t1 = MPI.Wtime()
        #comm.barrier()
        comm.send(split[i-1],dest = i)
        comm.send(totFreq,dest = i)
        comm.send(rows,dest = i)
        idf.extend(comm.recv(source= i))
        t2 = MPI.Wtime()
        time.append(t2-t1)
    print("Time taken is ",np.mean(time))
    print(idf[0:1])

else:
    partFile = comm.recv(source = 0)
    docFreq = comm.recv(source = 0)
    rows = comm.recv(source = 0)
    comm.send(idfCalc(partFile,docFreq,rows),dest = 0)

comm.barrier()
```

The function returns the same IDF for tokens, irrespective of the number of vectors, like 1.654 for ‘**rochester’** when 2,4 and 8 processes are considered. The ipython notebook has the complete IDF for one document for the three cases. The time required is,

Processes	Time
2	8.66754
3	6.38699
4	5.63123

This process require more time than term frequency as we have to go through all the documents at the beginning to create a dictionary that can be reference by all workers, a sample output is

[{'rochester': 1.6540237950243666, 'rutgers': 2.8760230272452265, 'usc': 2.1527228085145937, 'mpye': 9.903337541285003, 'media': 3.9346299812996373, 'horrified': 7.823895999605167, 'perot': 6.6452410032635205, 'investigating': 6.189765474580695, 'bush': 4.645842169257222, 'jun': 6.812295087926687, 'b': 0.05647325168996548, 'california': 3.322698404000054, 'state': 2.269967891605419, 'polytechnic': 5.249377191127479, 'university': 1.1670088199520945, 'lance': 5.776203156239911, 'visser': 6.347989479795589, 'writes': 0.6502250774649513, 'ca': 2.074503013696914, 'find': 2.189106396435917, 'retract': 6.812295087926687, 'claim': 3.263461707458467, 'go': 2.0320263379615966, 'dig': 5.8779858505498535, 'one': 1.0432641169522223, 'library': 4.250848361016352, 'fair': 4.290209434896933, 'enough': 2.536892392957404, 'serious': 3.6967616145600757, 'talking': 3.292641496567244, 'retracting': 9.903337541285003, 'anything': 2.4234734101199766, 'produced': 4.815741206052619, 'top': 3.4639871701849043, 'head': 3.5700579131453125, 'assume': 3.513096874219653, 'memory': 3.5472298805891116, 'source': 3.317165886430328, 'without': 2.3474324476736563, 'conditions': 4.727187808711173, 'substantiate': 6.684461716416802, 'statement': 3.540309437744538, 'rudman': 9.903337541285003, 'says': 2.621263883191538, 'believe': 2.2277915387471547, 'point': 2.244637983016704, 'provided': 4.098202572368514, 'produce': 4.323607715298781, 'give': 2.4336833683528747, 'reason': 2.7510686852524637, 'trouble': 3.9737483978951085, 'agree': 3.0724633066388236, 'simple':

3.347980649474338, 'know': 1.290470599800483, 'waste': 4.422698617943012, 'blow': 4.990682655548951, 'tripe': 7.418430891497002}]

Exercise 4 : Calculate Term Frequency Inverse Document Frequency

The TF-IDF is the product of the term frequency and inverse document frequency of each token respectively. This can be easily done as the foundations are laid in the previous two exercises, The root does the pre-processing required for computing IDF and send the data to the workers; it also send the split of documents to the workers for which they are calculating the TF-IDF. The function used is,

```
def tfidfCalc(arrayIdf,arrayTfid,arrayTotal,row):
    final = []
    #print("ArrayIdf",arrayIdf)
    #print("arrayTfid",arrayTfid)
    for each,every in zip(arrayIdf,arrayTfid):
        new = Counter(list(every))
        tfid = {k: np.log(v*(row/arrayTotal[k])) for k, v in new.items()}
        for other in each:
            partialIdf = (Counter(str(other).split(',')))
            totalIdf = len((str(other).split(',')))
            idf = {k: v / totalIdf for k, v in partialIdf.items()}
            final.append({k : v*tfid[k] for k,v in idf.items()})
            #print(idf,"\n\n",tfid)
    return(final)
```

The code returns the same values irrespective to the number of processes, the sample output and times are

```
Length 19996
{'ogicse': 0.02438193283226492, 'cos': 0.2286761273227808, 'bob': 0.05927197239596147, 'blackshaw': 0.10205285063231
759, 'b': 0.010346244584421157, 'subject': 0.016841345221392193, 'damn': 0.03338861415094335, 'ferigner': 0.06501559
679515352, 'taken': 0.025385314378350808, 'apr': 0.0006370153635510577, 'corporation': 0.02886141016618791, 'open':
0.02548147000953925, 'peter': 0.056347044969976855, 'nelson': 0.07862209694929219, 'writes': 0.01489065062896835, 'a
rticle': 0.0060469827545193795, 'norway': 0.046371726296151966, 'appear': 0.031238141590139712, 'posting': 0.0244005
7408154394, 'place': 0.02155199268215582, 'although': 0.0247931083276134, 'always': 0.021096772559695293, 'escaped':
0.0407556438917341, 'understanding': 0.03193149438685184, 'appeal': 0.0420525831039127, 'allegedly': 0.0486792138677
0108, 'rational': 0.03957892114496113, 'people': 0.011705991891354768, 'scheme': 0.03630414553031717, 'might': 0.033
47609492799696, 'gives': 0.028067182228988834, 'king': 0.03154119403171746, 'olav': 0.06501559679515352, 'v': 0.0337
2931761675074, 'whoever': 0.03964738427460052, 'atlas': 0.047438611352450887, 'right': 0.015276488511515689, 'specia
l': 0.027025262883546092, 'status': 0.034460044362354975, 'title': 0.032975887271017586, 'based': 0.0235618447380918
67, 'mere': 0.04168462812820371, 'accident': 0.038096814164495806, 'birth': 0.03898890836403253, 'begin': 0.03341896
6428650526, 'quite': 0.021054514551815506, 'inexpensive': 0.04316673510866904, 'compared': 0.03261164701880639, 'si
x': 0.03312074148062832, 'former': 0.06229452894354174, 'presidents': 0.15818943361216056, 'still': 0.016158669731727412, 'alive': 0.03360365901328455, 'drawing': 0.036856211649245615, 'pensions': 0.14061359329351233, 'accounts': 0.038801954012824165, 'secret': 0.031307016533648396, 'service': 0.10847181143725995, 'maybe': 0.021157549853573217,
'president': 0.02863057134794579, 'corporate': 0.04059824475278191, 'world': 0.019114041663768313, 'sop': 0.07030679
664675617, 'retiring': 0.05972439694355089, 'senior': 0.041163954463879035, 'executives': 0.06331221090725879, 'give
n': 0.0220635248217204, 'etc': 0.01969592994917201, 'point': 0.017134641091730563, 'performed': 0.03850019188491092
5, 'part': 0.019095389027031857, 'compensation': 0.04963465770704463, 'package': 0.03058155476067376, 'royals': 0.04
938435371602944, 'perform': 0.037000700274325914, 'free': 0.0220635248217204, 'ride': 0.03344943987345615, 'better':
0.01778478075220651, 'done': 0.02093505064595957, 'ps': 0.03898890836403253, 'say': 0.014297160146217697, 'provide
d': 0.03128398910204972, 'country': 0.025766256669179026, 'dissimilar': 0.05802101105565616, 'occurs': 0.03755519785
936387, 'bull': 0.04385079738874298, 'services': 0.026073691013743454, 'cow': 0.0501613208481664, 'cattle': 0.053970
413719303716, 'breeding': 0.052729811204053526}
```

Processes	Time
2	9.34579
3	7.28880
4	6.13477