Distributed Data Analytics Lab

Exercise Sheet 2

Sebastián Pineda Arango Matrikel-Nr: 246098

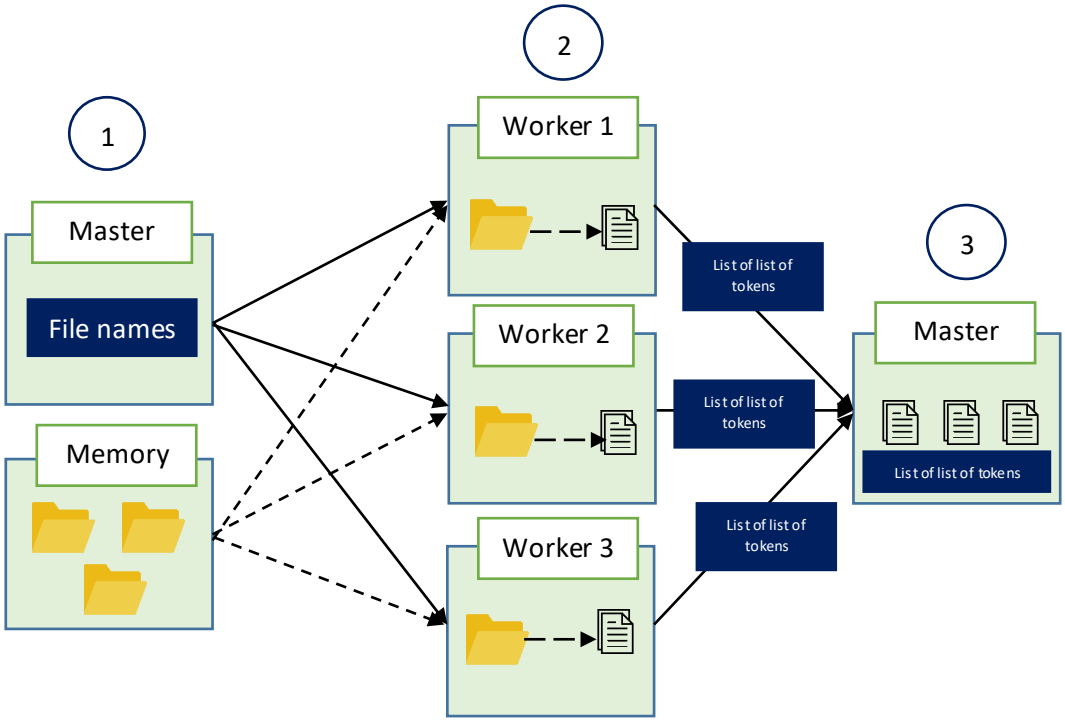
In this lab we are going to implement an algorithm t calculate the TF-IDF [1] matrix of a set of documents but in a distributed setting using MPI in Python. The laboratory is divided throughout different exercises. The dataset used for this task is 20NewsGroups dataset, which is available in [2].

Exercise 1: Data Cleaning and Text Tokenization

The first part of this lab aims to read the files: 20 folders with 1000 different texts. Then, the text in the files should be cleaned and tokenized. Now we explain the followed strategy for performing this task in a distributed setting:

1. **File name distribution:** The **master** reads all the folders names and assigns different folders names to every worker. This is done using point-to-point communication (*comm.send(),comm.recv()*).
2. **Text tokenization, cleaning and stemming:** Every **worker** reads the files in the assigned folder. For every file, it applies: cleaning and tokenization. The cleaning task includes removing punctuation signs, non-alphabetic characters and stop words. For the cleaning and tokenization, the library NLTK was used. Some useful insights about this libraries were taken from [3]. At the final, also a stemming step is run. This will transform the tokens to the respective stem.
3. **Results collection:** Every **worker** returns a list of list containing the tokens. Every list is the group of tokens associated to a document. The **master** should concatenate this list of tokens to have a final result.

The previous steps are shown in the following figure.



- **Steps in code:**

1. **File distribution**

```
if worker == 0:

    #the master sends the folders name to every worker

    folders = os.listdir(path) #path pointing to the directories
    frac = int(len(folders)/(num_workers-1))

    i = 0
    for i in range(1, num_workers-1):
        comm.send(folders[((i-1)*frac):(i*frac)], dest=i)

    comm.send(folders[(i*frac):], dest=(i+1))
else:

    #every worker receives the folder list
```

```
    folders = comm.recv()
```

2. **Text tokenization, cleaning and stemming**

```
else:

    #every worker reads the group of folders and perform preprocessing.
    #For the preprocessing, the no-alphabetic signs are removied and then
    #the text is tokenized.

    news = read_news(folders) #reading news
    porter = PorterStemmer() #object for stemming
    en_stop = get_stop_words('en') #list of stop words

    docs = news.Text.apply(lambda x: preprocess_text(x, en_stop, porter))
#going through all files in the directory
    comm.send(docs, dest=0 ) #sending back data to the master
```

3. **Results collection**

```
if worker == 0:

    #the master receives back the list of tokens
    #and append every list in a list of lists

    docs = []
    for i in range(1, num_workers):
        docs_ = comm.recv(source=i)
        docs.append(docs_)

    final = MPI.Wtime()
    print("Elapsed time : ", final-init)
```

In the following table, we show the results for the timing while we increase the number of processes. The experiment is executed several times, so that we can have a more robust conclusion about the timing: in fact, the execution time decreases as the number of processes increases.

Number of processes	P=2	P=3	P=4	P=5
Time in 1st experiment	399.65 sec.	163.69 sec.	130.71 sec.	118.21 sec.
Time in 2nd experiment	534.16 sec.	236.33 sec.	191.11 sec.	181.95 sec.
Time in 3rd experiment	199.43 sec.	134.26 sec.	136.46 sec.	116.30 sec.
Time in 4th experiement	324.00 sec.	148.26 sec.	119.27 sec.	118.62 sec.

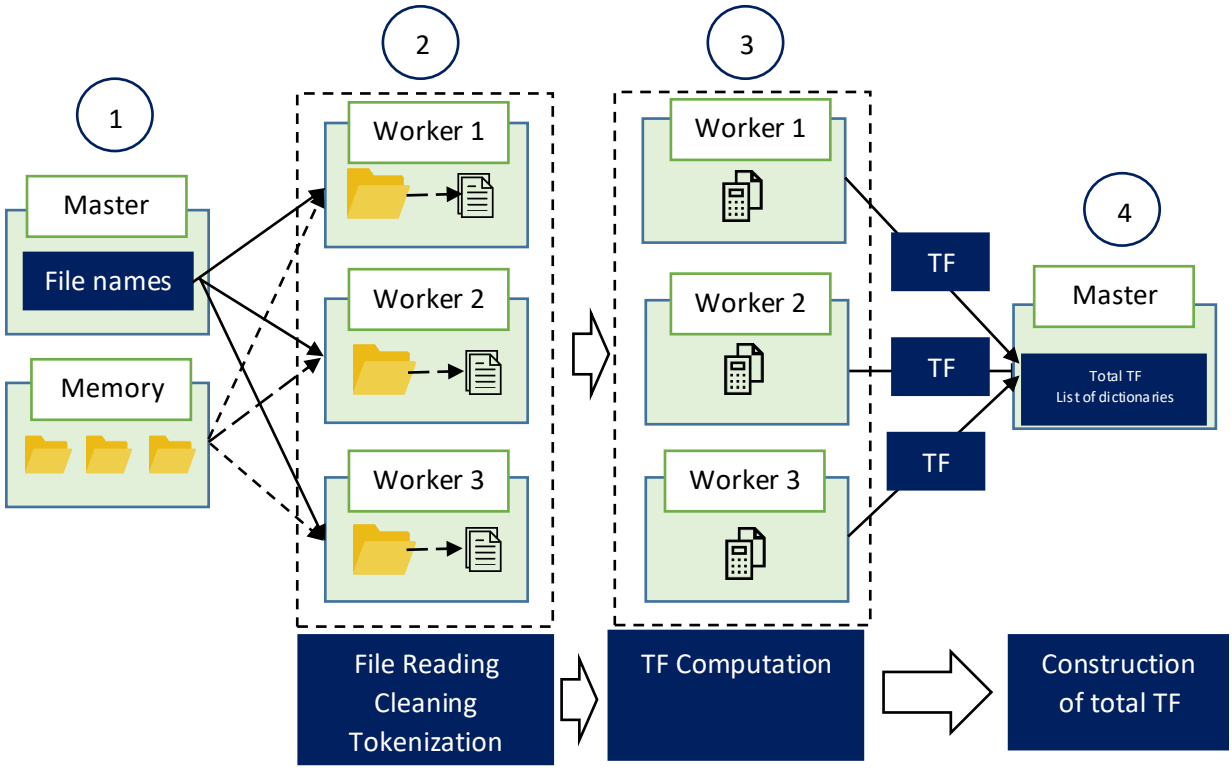
Exercise 2: Calculate Term Frequency (TF)

In the second exercise, we add a new task to the previous pipeline. We want to count now the frequency of every term in the texts that the workers have. This process is divided in the following steps using point-to-point communication:

1. **File name distribution:** The **master** reads all the folders names and assigns different folders names to every worker. This is done using point-to-point communication (*comm.send(),comm.recv()*).
2. **Text tokenization, cleaning and stemming:** Every **worker** reads the files in the assigned folder. For every file, it applies: cleaning and tokenization. The cleaning task includes removing punctuation signs, non-alphabetic characters and stop words. For the cleaning and tokenization, the library NLTK was used. Some useful insights about this libraries were taken from [3]. At the final, also a stemming step is run. This will transform the tokens to the respective stem.
3. **Term frequency computation:** Every **worker** counts the number of occurrences of every term in a document and divides this value by the total number of words: this is the term frequency. The term frequencies values are stored in a dictionary, since the data is sparse. The term frequency is given by the following equation:

$$\mathrm{TF}(t,d) = \frac{n^d(t)}{\sum_{t' \in d} n^d(t')}$$

4. **Term frequency results collection:** The **workers** send back the dictionary with the TF to the **master**. The **master** collect these dictionaries in a list. The results are stored in a JSON file.



- **Steps in code:**

1. **File name distribution**

```
if worker == 0:

    #the master sends the folders name to every worker

    folders = os.listdir(path)
    frac = int(len(folders)/(num_workers-1))

    i = 0
    for i in range(1, num_workers-1):
        comm.send(folders[((i-1)*frac):(i*frac)], dest=i)

    comm.send(folders[(i*frac):], dest=(i+1))
else:
    folders = comm.recv()
```

2. **Text tokenization, cleaning and stemming**

```
#text preprocessing: tokenization, cleaning and stemming
news = read_news(folders)
porter = PorterStemmer()
en_stop = get_stop_words('en')

docs = news.Text.apply(lambda x: preprocess_text(x, en_stop, porter))
```

3. **Term frequency computation**

```
#term frequency computation
tf_list = []
for text in docs:
    tf_worker= count_words(text)
    tf_list.append(tf_worker)

#sending the results to the master
comm.send(tf_list, dest=0 )
```

4. **Term frequency results collection**

```
tf_list = []
for i in range(1, num_workers):
    tf_list_ = comm.recv(source=i)
    tf_list.append(tf_list_)

final = MPI.Wtime()
print("Elapsed time : ", final-init)

with open("tf_lists", 'w') as fp:
    json.dump(tf_list,fp)
```

In the following table, we show the results for the timing while we increase the number of processes. The experiment is executed several times, so that we can have a more robust conclusion about the timing: in fact, the execution time decreases as the number of processes increases.

Number of processes	P=2	P=3	P=4	P=5
Time in 1st experiment	198.06 sec.	147.11 sec.	149.83 sec.	137.20 sec.
Time in 2nd experiment	234.11 sec.	235.96 sec.	157.93 sec.	127.59 sec.
Time in 3rd experiment	227.46 sec.	151.21 sec.	149.34 sec.	132.38 sec.
Time in 4th experiment	185.69 sec.	171.69 sec.	200.98 sec.	157.19 sec.

Exercise 3: Calculate Inverse Document Frequency (IDF)

In this third exercise, the pipeline grows since we add the calculation of the Inverse Document Frequency (IDF), which is given by the following equation:

$$\mathrm{IDF}(t) = \log \frac{|C|}{\sum_{d \in C} \mathbb{1}(t, d)}$$

To compute this entirely, every worker must count among how many of the assigned documents contains every different word (token). This value is denoted as DF in the graph below. With this value, the master is able to combine the results to calculate the total IDF for every word. Summarizing, we perform the following steps:

1. **File name distribution:** The **master** reads all the folders names and assigns different folders names to every worker. This is done using point-to-point communication (*comm.send(),comm.recv()*).
2. **Text tokenization, cleaning and stemming:** Every **worker** reads the files in the assigned folder. For every file, it applies: cleaning and tokenization. The cleaning task includes removing punctuation signs, non-alphabetic characters and stop words. For the cleaning and tokenization, the library NLTK was used. Some useful insights about this libraries were taken from [3]. At the final, also a stemming step is run. This will transform the tokens to the respective stem.

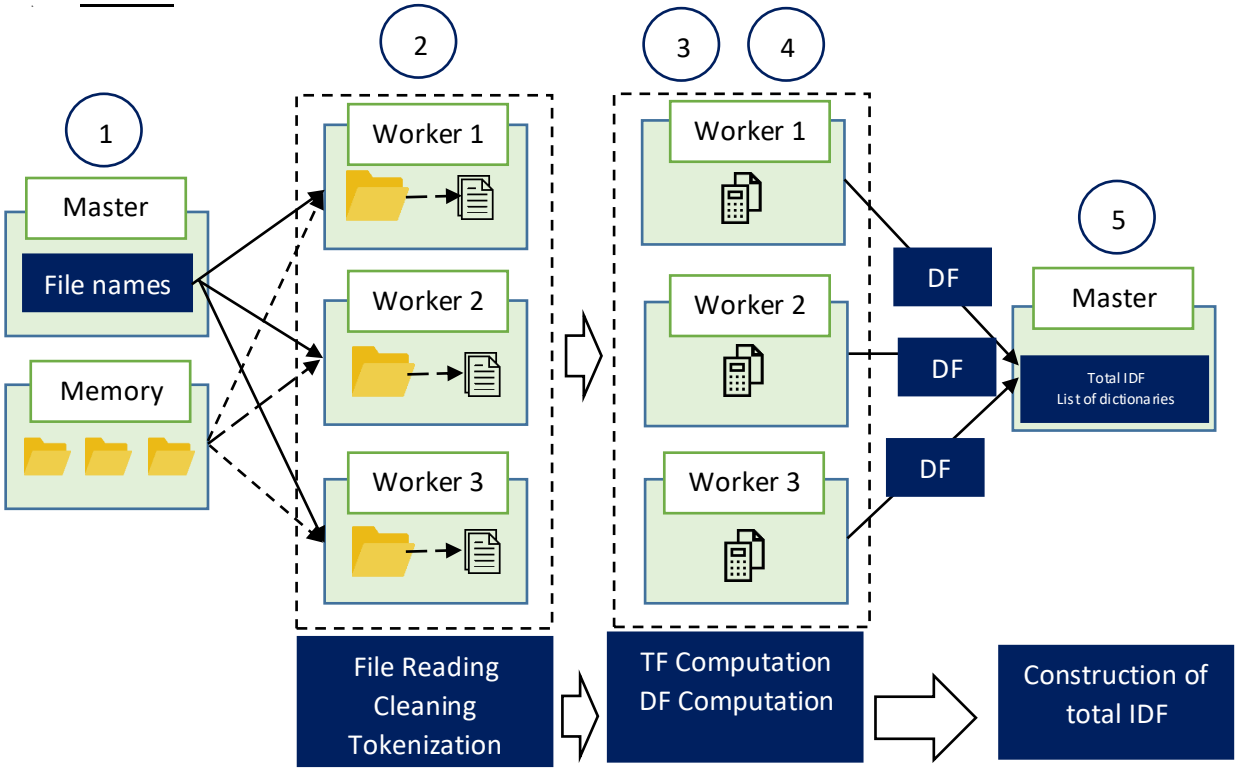
3. **Term frequency computation:** Every **worker** counts the number of occurrences of every term in a document and divides this value by the total number of words: this is the term frequency. The term frequencies values are stored in a dictionary, since the data is sparse.
4. **Computation of DF:** every worker counts how many times every work appears among the documents it has (DF):

$$DF_i(t) = \sum_{d \in W_i}^{N_i} I(t,d)$$

Where DF_i corresponds to the value calculated by the worker i (W_i) and N_i is the number of documents of the worker i.

5. **DF results collection and final IDF calculation:** every worker sends back the calculated DF as a dictionary. The master add the results of the dictionaries so that the values of common keys among the dictionaries are added: it is, the master computes the intersection of all the DF. Finally, it computes the final IDF such as:

$$IDF(t) = \log\left(\frac{|C|}{\sum_i^{|W|} DF_i}\right)$$



- **Steps in code:**

1. **File name distribution**

```
if worker == 0:

    #the master sends the folders name to every worker

    folders = os.listdir(path)
    frac = int(len(folders)/(num_workers-1))

    i = 0
    for i in range(1, num_workers-1):
        comm.send(folders[((i-1)*frac):(i*frac)], dest=i)

    comm.send(folders[(i*frac):], dest=(i+1))
else:
    folders = comm.recv()2
```

2. **Text tokenization, cleaning and stemming**

```
else:
    #reading files
    news = read_news(folders)

    #preprocessing
    porter = PorterStemmer()
    en_stop = get_stop_words('en')

    docs = news.Text.apply(lambda x: preprocess_text(x, en_stop, porter))
```

3. **Term frequency computation**

```
  #computing term frequency
    tf_list = []
    for text in docs:

        tf_worker= count_words(text)
        tf_list.append(tf_worker)
```

4. **Computation of DF**

```
#computing DF (for IDF)
    idf_dict = {}
    for tf in tf_list:
        for key in tf.keys():
            if key in idf_dict.keys():
                idf_dict[key]+=1
            else:
                idf_dict[key]=1

    #sending information to master
    comm.send({"tf": tf_list,
               "idf": idf_dict}, dest=0 )
```

5. **DF results collection and final IDF calculation**

```
if worker == 0:

    #concatenation of the term frequency dictionaries sent by the workers
    #and addition of the dictionaries with DF

    tf_list = []
    idf_dict = {}
    for i in range(1, num_workers):
        tfidf = comm.recv(source=i)

        tf_list_ = tfidf["tf"]
        idf_dict_ = tfidf["idf"]

        tf_list.append(tf_list_)
        idf_dict = add_dict(idf_dict, idf_dict_)

    n_docs = len(tf_list)

    #calculation of the final IDF for all the terms
    for key in idf_dict.keys():
```

```
        idf_dict[key] = np.log(n_docs/idf_dict[key])
```

In the following table, we show the results for the timing while we increase the number of processes. The experiment is executed several times, so that we can have a more robust conclusion about the timing: in fact, the execution time decreases as the number of processes increases.

Number of processes	P=2	P=3	P=4	P=5
Time in 1st experment	208.88 sec.	171.84 sec.	150.21 sec.	137.35 sec.
Time in 2nd experiment	232.14 sec.	236.38 sec.	145.23 sec.	246.24 sec.
Time in 3rd experiment	216.13 sec.	158.94 sec.	188.05 sec.	171.83 sec.
Time in 4th experiment	215.88 sec.	180.25 sec.	148.63 sec.	148.05 sec.

Exercise 4: Calculate Term Frequency Inverse Document Frequency (TF-IDF)

In the last part, we calculate the TF-IDF value of every term in every document. To do that, we simply add a final step to the previous pipeline. The new stage consist in a collective communication of the master to all the workers: the master broadcast the IDF of all terms. Then the workers can compute the TF-IDF values associated to the documents they have. The total steps involved in the pipline are:

1. **File name distribution:** The **master** reads all the folders names and assigns different folders names to every worker. This is done using point-to-point communication (*comm.send(),comm.recv()*).
2. **Text tokenization, cleaning and stemming:** Every **worker** reads the files in the assigned folder. For every file, it applies: cleaning and tokenization. The cleaning task includes removing punctuation signs, non-alphabetic characters and stop words. For the cleaning and tokenization, the library NLTK was used. Some useful insights about this libraries were taken from [3]. At the final, also a stemming step is run. This will transform the tokens to the respective stem.
3. **Term frequency computation:** Every **worker** counts the number of occurrences of every term in a document and divides this value by the total number of words: this is the term frequency. The term frequencies values are stored in a dictionary, since the data is sparse.
4. **Computation of DF:** every worker counts how many times every work appears among the documents it has (DF):

$$DF_i(t) = \sum_{d \in W_i}^{N_i} I(t,d)$$

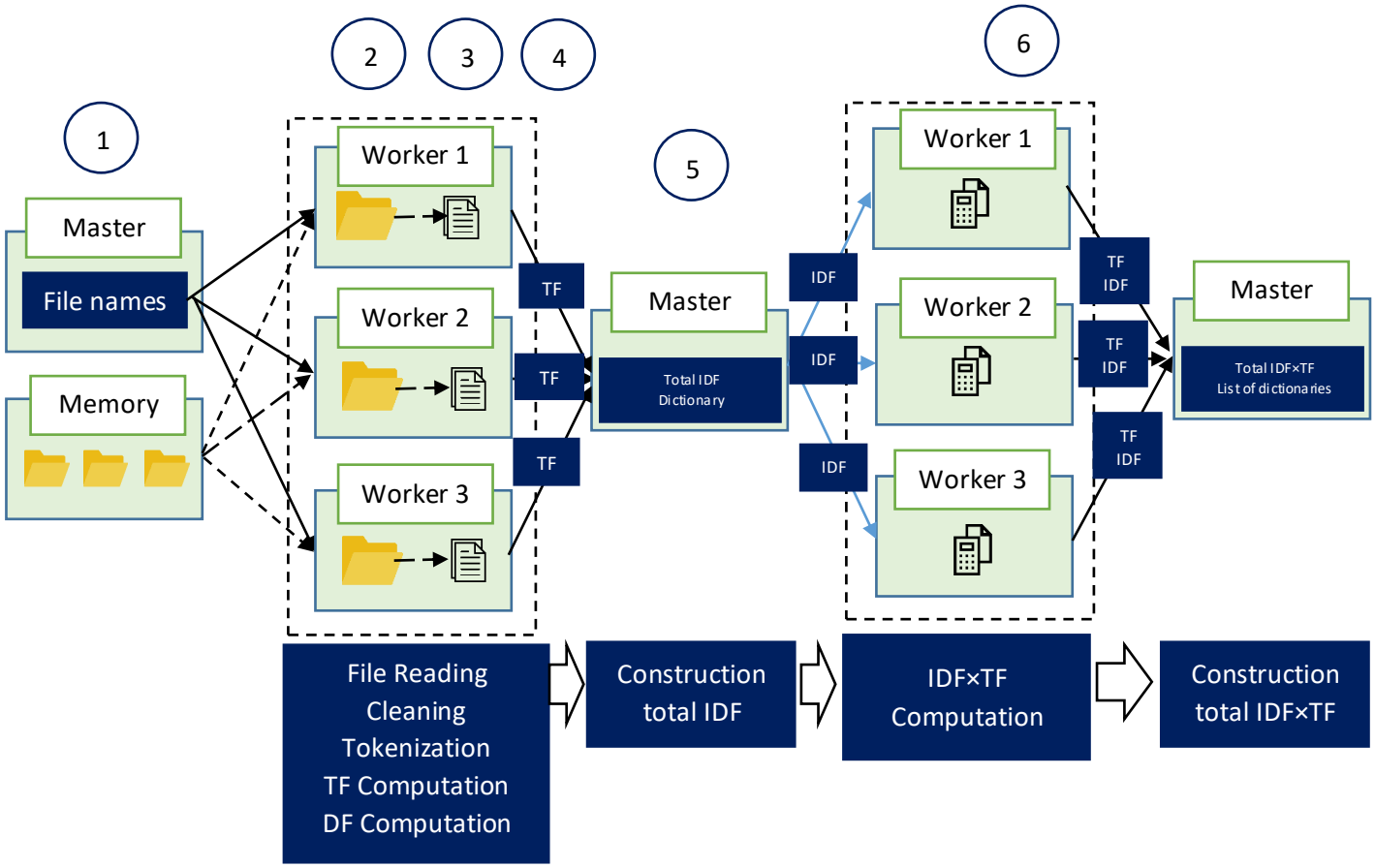
Where DF_i corresponds to the value calculated by the worker i (W_i) and N_i is the number of documents of the worker i.

5. **DF results collection IDF calculation:** every worker sends back the calculated DF as a dictionary. The master add the results of the dictionaries so that the values of common keys among the dictionaries are added: it is, the master computes the intersection of all the DF. Finally, it computes the final IDF such as:

$$IDF(t) = \log(\frac{|C|}{\sum_i^{|W|} DF_i})$$

6. **TF×IDF calculation:** every worker receives the full IDF-dictionary through a **broadcast** operation of the master. Then, they use this IDF-dictionary to compute the final TF-IDF values of the assigned documents.

7. **Final TF-IDF construction.** The final TF-IDF values are sent back to the master, which combines the result in a list of dictionaries outputs the result as a JSON file.



1. **File name distribution**

```
if worker == 0:

    folders = os.listdir(path)
    frac = int(len(folders)/(num_workers-1))

    i = 0
    for i in range(1, num_workers-1):
        comm.send(folders[((i-1)*frac):(i*frac)], dest=i)

    comm.send(folders[(i*frac):], dest=(i+1))
else:
    folders = comm.recv()
```

2. **Text tokenization, cleaning and stemming**

```
    #preprocessing
    news = read_news(folders)
    porter = PorterStemmer()
    en_stop = get_stop_words('en')

    docs = news.Text.apply(lambda x: preprocess_text(x, en_stop, porter))
```

3. **Term frequency computation**

```
    #computing term frequency
    tf_list = []
    for text in docs:

        tf_worker= count_words(text)
        tf_list.append(tf_worker)
```

4. **Computation of DF**

```
    #computing document frequency
    idf_dict = {}
    for tf in tf_list:
        for key in tf.keys():
            if key in idf_dict.keys():
                idf_dict[key]+=1
            else:
                idf_dict[key]=1


    #sending back resultss
    comm.send({"n_docs": len(tf_list),
               "idf": idf_dict}, dest=0 )
```

5. **DF results collection and IDF calculation**

```
if worker == 0:

    #concatenation of the term frequency dictionaries sent by the workers
    #and addition of the dictionaries with DF
    n_docs = 0
    idf_dict_cpt = dict()
    for i in range(1, num_workers):
        tfidf = comm.recv(source=i)
        n_docs_ = tfidf["n_docs"]
        idf_dict_ = tfidf["idf"]
        n_docs += n_docs_
        idf_dict_cpt = add_dict(idf_dict_cpt, idf_dict_)

    #computation of the final IDF
    for key in idf_dict_cpt.keys():
        idf_dict_cpt[key] = np.log(n_docs/idf_dict_cpt[key])

#broadcasting the final idf dictionary
idf_dict_cpt = comm.bcast(idf_dict_cpt, root=0)
```

6. **TF×IDF calculation**

```
#computing the final tf-idf for the assigned documents
for tf_doc in tf_list:
    for key in tf_doc.keys():
        tf_doc[key] = tf_doc[key]*idf_dict_cpt[key]
```

7. **Final TF-IDF construction**

```
#combining the tf-idf received from workers
tfidf_list = []
for i in range(1, num_workers):
    tfidf = comm.recv(source=i)
    tfidf_list.append(tfidf)
with open("tfidf", 'w') as fp:
    json.dump(tfidf_list,fp)
```

The final generated JSON file looks like the following image:

[{"tfidf": [{"clau": 0.007153742421887177, "platinga":
0.008881572189705937, "isbn": 0.02879224642507559, "michael":
0.0028724446916455577, "although": 0.006013018122038714, "martin":
0.00875997179474557, "chill": 0.006094328450172511, "spent":
0.004210224951076631, "miz": 0.008881572189705937, "coher":
0.005992727181457152, "noteworthi": 0.0067627442462766035,
"anoth": 0.0021345312175009828, "gott": 0.008213156393601845,
"contact": 0.003243733448088075, "planet": 0.0041766475497659,
"defenc": 0.005554679667206818, "grow": 0.0042851996404444015,
"second": 0.002535729241504444, "hunt": 0.0050260266721850145,
"publish": 0.01438592647013719, "delight": 0.006042190339015 06,
"outstand": 0.006121494194595554, "music": 0.013167249974374937,
"germani": 0.014626307432140113, "hardcov": 0.014010183629064117,
"franc": 0.004718131385910075, "ball": 0.007962943096896036,
"promot": 0.004455288150961441, "black": 0.006611195418670998,
"agnost": 0.005425912654068416, "de": 0.010379882950133082,
"cathol": 0.004416902181433553, "saint": 0.004841403682448055,
"often": 0.008999980078536723, "al": 0.0036086115907328123,
"cleric": 0.007153742421887177, "oxford": 0.01069745171570957,
"date": 0.0, "realiti": 0.003809155455156431, "john":
0.0024957754595294392, "hall": 0.0041766475497659, "work":
0.004682490688798171, "day": 0.0020705324950211043,

The execution time for different number of processes is registered in the following table. Here, we don't see a clear difference among the different settings. We hypothesize that this is due to the communication overload generated by the transfer of the whole IDF dictionary.

Number of processes	P=2	P=3	P=4	P=5
Time in 1st experiment	228.75 sec.	289.274 sec.	796.30 sec.	716.78 sec.
Time in 2nd experiment	370.25 sec.	217.48 sec.	238.77 sec.	216.54 sec.
Time in 3rd experiment	268.32 sec.	262.05 sec.	237.15 sec.	143.63 sec.
Time in 4th experiment	238.20 sec.	262.08 sec.	240.58 sec.	219.64 sec.

Technical specifications note:

This lab uses several libraries for Python listed as follows:

- Numpy
- Os
- MPI4PY
- Time
- Json
- NLTK

For NLTK we also download an extra package called "punkt".

```
install  python -m nltk.downloader punkt
```

Function definition note:

In the above mentioned algorithms, some functions where used, here they are specified.

```
#defining functions
def add_dict(dict1, dict2):

    '''This function adds two dictionaries so that the common values
    of the common keys are added and the disjunct keys (keys that only
    one of the dictionaries has) are kept.'''

    for key1 in dict1:
        if key1 in dict2:
            dict2[key1]+=dict1[key1]
        else:
            dict2[key1]=dict1[key1]
    return dict2


def read_news(folders):

    '''Function to read news located in given path folder'''

    class_list = []
    text_list = []

    for folder in folders:
        files = os.listdir(path+"/"+folder)

        for file_name in files:
            file = open(path+"/"+folder+"/"+file_name, 'r')
            text = file.read()

            class_list.append(folder)
            text_list.append(text)

    data = pd.DataFrame({'Text': text_list,
                         'Label': class_list})

    return data


def preprocess_text (x, stop_words, porter):
```

```
    '''This function preprocess a string, eliminateing sop words and
    non-alphanumeric characters.'''

    x = x.lower()
    tokens = word_tokenize(x)
    words = [word for word in tokens if word.isalpha()]
    words = [w for w in words if not w in stop_words]
    x = [porter.stem(word) for word in words]

    return x



def count_words (text):

    '''This function counts the number of ocurrences for every word in the
text'''

    #number of words
    n_words  = len(text)

    #unique workds
    unique_words = set(text)

    #creating dictionary to count
    dict_count= {}

    #8. counting words
    for w in unique_words:
        dict_count[w]=text.count(w)/n_words


    #getting the list of keys
    keys = list(dict_count.keys())

    #getting the list of values
    values = list(dict_count.values())

    #creating dataframe with keys and values as columns
    df_count = dict(zip(keys, values))


    return df_count
```

References:

[1] TF-IDF: https://es.wikipedia.org/wiki/Tf-idf

[2] 20 New groups datasets: http://qwone.com/~jason/20Newsgroups/

[3] https://machinelearningmastery.com/clean-text-machine-learning-python/

[4] Icons for figures taken from: www.flaticon.com