

## Test report for Project 2

```
# necessary imports for the notebook to work
import os
import re
import math
from collections import defaultdict
import numpy as np
import random
```

```
# Set input file path and output directory
input_file_path = 'cran-1.all.1400'
output_directory = 'split_documents'

# Split documents
split_documents(input_file_path, output_directory)

# Set input directory and N
input_directory = 'split_documents'
N = 1400
```

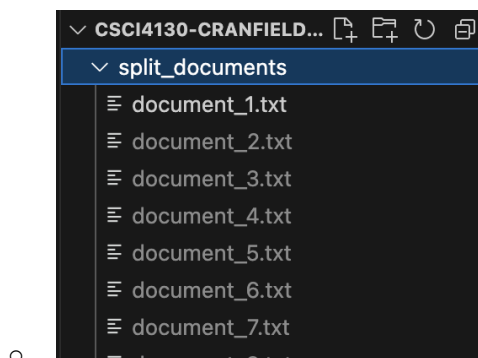
With the initial variables setting like above, the functions operates successfully.

### Document Splitting Function ('split\_documents')

The split\_documents function serves as the initial step in our document processing pipeline. Below is an overview of its functionality:

1. **Purpose:** This function is designed to split a large input file into individual documents and save each document in a separate file within a specified directory.
2. **Arguments:**
  - **input\_file\_path** (str): Path to the input file containing multiple documents.
  - **output\_dir** (str): Directory where the split documents will be saved.
3. **Implementation:**
  - It first checks if the output directory exists. If not, it creates the directory.
  - The input file is then opened and its content is read.
  - Documents are split based on a specific pattern, typically identified by a marker such as ".I" followed by a number.
  - For each split document, text content following the ".W" marker is extracted using regular expressions.
  - The extracted text is then saved into separate files within the output directory, with filenames structured as "document\_{i}.txt", where {i} represents the index of the document.

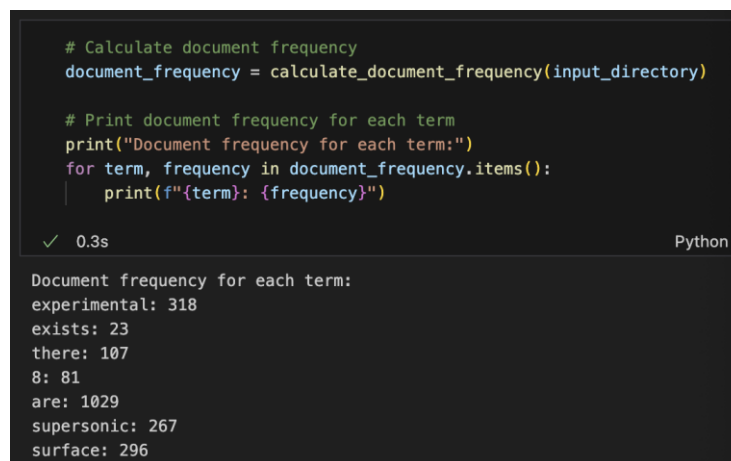
### 4. Result



## Document Frequency Calculation Function (`calculate_document_frequency`)

The `calculate_document_frequency` function computes the document frequency of terms within a collection of documents. Here's an overview:

1. **Purpose:** It aims to generate a dictionary containing the document frequency of terms across the entire document collection.
2. **Arguments:**
  - `input_dir` (str): Directory containing the individual documents.
3. **Implementation:**
  - It initializes a defaultdict to store the document frequency.
  - For each document file in the input directory, it reads the content and converts it to lowercase.
  - Using regular expressions, it identifies unique terms within each document.
  - The frequency of each term is incremented in the document frequency dictionary accordingly.
4. **Returns:**
  - A dictionary containing the document frequency of terms.



```
# Calculate document frequency
document_frequency = calculate_document_frequency(input_directory)

# Print document frequency for each term
print("Document frequency for each term:")
for term, frequency in document_frequency.items():
    print(f"{term}: {frequency}")
```

✓ 0.3s Python

Document frequency for each term:  
experimental: 318  
exists: 23  
there: 107  
8: 81  
are: 1029  
supersonic: 267  
surface: 296

## Term Frequency Calculation Function (`calculate_tf`)

The `calculate_tf` function computes the term frequency for each term in each document. Here's a summary:

1. **Purpose:** It aims to create a nested dictionary containing the term frequency for each term in each document.
2. **Arguments:**
  - `document_directory` (str): Directory containing the individual documents.
3. **Implementation:**
  - It initializes a defaultdict of dictionaries to store the term frequency.
  - For each document file in the document directory, it reads the content and converts it to lowercase.

- Using regular expressions, it identifies unique terms within each document and calculates their frequency.
- The term frequency for each term in each document is stored in the nested dictionary.

#### 4. Returns:

- A nested dictionary containing the term frequency for each term in each document.

```
# Calculate term frequency
tf = calculate_tf(input_directory)

# Print term frequency for each term in each document
print("\nTerm frequency for each term in each document:")
for term, doc_tf in tf.items():
    print(f"{term}: {doc_tf}")
```

✓ 0.6s Python

```
Term frequency for each term in each document:
a: {1: 7, 2: 9, 3: 1, 4: 4, 5: 4, 6: 4, 7: 1, 8: 5, 9: 9, 10: 1, 11: 1,
factor: {24: 1, 55: 1, 63: 1, 81: 1, 82: 1, 162: 1, 238: 1, 253: 1, 274:
affecting: {82: 1, 173: 1, 189: 2, 215: 1, 272: 1, 439: 2, 440: 1, 710:
transonic: {38: 2, 118: 2, 121: 1, 124: 2, 157: 1, 197: 4, 214: 1, 216:
leading: {2: 1, 8: 1, 9: 1, 25: 3, 26: 3, 39: 1, 105: 1, 134: 1, 146: 2,
```

### Inverse Document Frequency Calculation Function (calculate\_idf\_t)

The **calculate\_idf\_t** function calculates the inverse document frequency for each term.

Here's a brief explanation:

1. **Purpose:** It aims to compute the inverse document frequency for each term across the entire document collection.
2. **Arguments:**
  - o **N** (int): Total number of documents in the collection.
  - o **document\_frequency** (dict): Dictionary containing the document frequency of terms.
3. **Implementation:**
  - o It initializes an empty dictionary to store the inverse document frequency.
  - o For each term in the document frequency dictionary, it computes the inverse document frequency using the formula:  $IDF = \log\left(\frac{N}{DF_t}\right)$ , where  $DF_t$  represents the document frequency of the term.
  - o If the document frequency of a term is zero, its IDF value is set to zero to avoid division by zero errors.
4. **Returns:**
  - o A dictionary containing the inverse document frequency of terms.

```
# calculate all idf values
idf_t = calculate_idf_t(N, document_frequency)
# print 5 sample terms with their idf values
print("\nIDF for 5 sample terms:")
for term, doc_values in list(idf_t.items())[:5]:
    print(f"{term}: {doc_values}")
```

✓ 0.0s

IDF for 5 sample terms:  
 experimental: 1.482176132823173  
 exists: 4.1087332996742  
 there: 2.571398681141444  
 8: 2.849778360930911  
 are: 0.30788477976930034

### TF-IDF Calculation Function (calculate\_tf\_idf)

The **calculate\_tf\_idf** function computes the TF-IDF values for each term in each document. Here's a succinct overview:

1. **Purpose:** It aims to generate a nested dictionary containing the TF-IDF values for each term in each document.
2. **Arguments:**
  - **tf** (dict): Nested dictionary containing the term frequency for each term in each document.
  - **idf** (dict): Dictionary containing the inverse document frequency of terms.
3. **Implementation:**
  - It initializes a defaultdict of dictionaries to store the TF-IDF values.
  - For each term in the term frequency dictionary, it computes the TF-IDF value for each document using the formula:  $TF-IDF = TF \times IDF$ .
  - The TF-IDF values are stored in the nested dictionary.
4. **Returns:**
  - A nested dictionary containing the TF-IDF values for each term in each document.

```
# Calculate TF-IDF values
tf_idf_values = calculate_tf_idf(tf, idf_t)

# print 5 sample terms with their tf-idf values per document
print("\nTF-IDF values for 5 sample terms.")
for term, doc_values in list(tf_idf_values.items())[:5]:
    print(f"{term}: {doc_values}")
```

✓ 3.2s

TF-IDF values for 5 sample terms.  
 a: {1: 0.4972504118172622, 2: 0.6393219580507656, 3: 0.07103577311  
 factor: {1: 0.0, 2: 0.0, 3: 0.0, 4: 0.0, 5: 0.0, 6: 0.0, 7: 0.0, 8  
 affecting: {1: 0.0, 2: 0.0, 3: 0.0, 4: 0.0, 5: 0.0, 6: 0.0, 7: 0.0  
 transonic: {1: 0.0, 2: 0.0, 3: 0.0, 4: 0.0, 5: 0.0, 6: 0.0, 7: 0.0  
 leading: {1: 0.0, 2: 2.473542891137685, 3: 0.0, 4: 0.0, 5: 0.0, 6:

## How our scoring works

The *score* function in our notebook takes a query string as well as the matrix of tf-idf values for each term in each document and scores them by counting up the sum of tf-idf values for all terms in the query across their documents. It gives back the top ten documents according to the tf-idf values, ranked descendingly. The *eval* function picks out queries which correspond to at least 15 documents according to the cranqrel file, in order to avoid precision calculations of zero due to none of the documents matching up. It then intersects those IDs with the IDs from the cran.qry file and randomly picks 20 IDs from that intersection. These are the random 20 queries we execute and they are printed when executing the cell containing the *calc\_prec* function. The following is a screenshot from one run of the notebook (bearing in mind that the precision and average precision can vary from run to run due to the stochasticity when randomly picking 20 queries to execute):

```
QueryIDs of queries ran:
dict_keys([201, 212, 132, 2, 157, 57, 156, 1, 67, 217, 225])
Precision scores for the ran queries:
[0.0, 0.1, 0.0, 0.4, 0.0, 0.0, 0.0, 0.6, 0.0, 0.0, 0.1]
Average precision of the system across all 20 queries for the current run:
0.10909090909090911
```

For instance, running query number 212 gives a precision of 0.1, meaning 1 of the top-10 scored documents was also found listed in the cranqrel file for queryID 212. Averaging the precision values for all 20 queries run gives a precision of 0.109