COVID-19 (report)

Team members

Richa Bandlas - 1913106
Paras Yadav - 170010007
Rajat Kumar Dalai - 170010003
Naresh Kumar Kaushal - 170030027
Bhavam Gupta - 170020029
Rahul Ratneshwar Mandal - 170010006

Table of Contents

Data Preprocessing

The data originally in JSON format and is too complex to use directly. So we have to convert into csv format, for that we needed some helper functions to easily format inner dictionaries from each file. Papers contain many sections and subsections which need to be formatted. Since multiple subsections can have the same section, we need to first group each subsection and then concatenate.

1. format_name(author) : Joins the names in sequence of <first><middle><last>.

```
def format_name(Authors)
    middle_name = " ".join(Authors['middle'])

    if Authors['middle']:
        return " ".join([Authors['first'], middle_name, Authors['last']])
    else:
        return " ".join([Authors['first'], Authors['last']])

```

2. format_affiliation(affiliation) : If institution and location details are there in json then put it in a list.

```python
def format_affiliation(affiliation):
    affiliate = []
    location = affiliation.get('location')
    if location:
        affiliate .extend(list(affiliation['location'].values()))

    institution = affiliation.get('institution')
    if institution:
        affiliate = [institution] + affiliate
    return ", ".join(affiliate )
```

3. format_authors() : Joins the author's name with the affiliation if it is available.

```python
def format_Authors(Authors, with_affiliation=False):
    name_ls = []

    for Author in Authors:
        name = format_name(Author)
        if with_affiliation:
            affiliation = format_affiliation(Author['affiliation'])
            if affiliation:
                name_ls.append(f"{name} ({affiliation})")
            else:
                name_ls.append(name)
        else:
            name_ls.append(name)

    return ", ".join(name_ls)
```

4. format_body() : Extracts the text and then append it into a list.
5. format_bib() : Joins the title, authors, venue, year together to form a string.

We worked to extract useful information from the biorxiv and the other datasets and convert them into a more readable comma seperated values (csv) format.

1. The JSON files in each directory are loaded and appended into a python list.
2. Through a series of system calls to the helper functions, relevant information is extracted from each article and stored in a list.
3. The feature list generated from each article is appended into another list named cleaned_files which, combined with the appropriate column names, gives us the clean dataframe.

Each row of the dataframe now represents an article. The dataframe is converted into a csv file via the pandas.DataFrame.to_csv python function.

```python
def generate_clean_df(all_files):
    cleaned_files = []
    for file in tqdm(all_files):
        features = [
            file['paper_id'],
            file['metadata']['title'],
            format_authors(file['metadata']['authors']),
            format_authors(file['metadata']['authors'], with_affiliation=Tru
            format_body(file['abstract']),
            format_body(file['body_text']),
            format_bib(file['bib_entries']),
            file['metadata']['authors'],
            file['bib_entries']
        ]
        cleaned_files.append(features)
    col_names = ['paper_id', 'title', 'authors','affiliations', 'abstract',
    clean_df = pd.DataFrame(cleaned_files, columns=col_names)
    return clean_df
```

This code generate csv files in 3 lines using the load_files and generate_clean_dr helper functions.

```
comm_dir = '/kaggle/input/CORD-19-research-challenge/comm_use_subset/comm_us
comm_files = load_files(comm_dir)
comm_df = generate_clean_df(comm_files)
comm_df.to_csv('clean_comm_use.csv', index=False)
comm_df.head()
```

Bag Of Words

The bag-of-words model is a simplifying representation used in natural language processing and information retrieval (IR). In this model, a text (such as a sentence or a document) is represented as the bag (multiset) of its words, disregarding grammar and even word order but keeping multiplicity.
The bag-of-words model is commonly used in methods of document classification where the (frequency of) occurrence of each word is used as a feature for training a classifier.
Here, i have loaded all csv files which we converted in preprocessing step. Then, null values in each dataframe is replaced with missing value. After this, we have merged all the csv files into one file called papers and then we are cleaning up the abstract and text data like removing new lines, citation number , figures and table etc.

```
bio = pd.read_csv("C:/Users/91869/Desktop/ML/Cord-19/biorxiv_clean.csv")
noncomm = pd.read_csv("C:/Users/91869/Desktop/ML/Cord-19/clean_noncomm_use.c
comm = pd.read_csv("C:/Users/91869/Desktop/ML/Cord-19/clean_comm_use.csv")
pmc = pd.read_csv("C:/Users/91869/Desktop/ML/Cord-19/clean_pmc.csv")

bio = bio.fillna("Missing")
noncomm = noncomm.fillna("Missing")
noncomm = noncomm.fillna("Missing")
comm = comm.fillna("Missing")
pmc = pmc.fillna("Missing")

papers = pd.concat([bio, comm, noncomm, pmc], ignore_index=True)

from nltk.tokenize import word_tokenize

print("The shape of our data:",papers.shape,"\n")

# print columns names
print("Our column names are:",papers.columns.values)

word_tokenize(papers['abstract'][0])

papers['abstract'][0]

def clean_up(t):
    """
    Cleans up the passed value
    """
    # Remove New Lines
    t = t.replace("\n"," ") # removes newlines

    # Remove citation numbers (Eg.: [4])
    t = re.sub("\[[0-9]+(, [0-9]+)*\]", "", t)

    # Remove et al.
    t = re.sub("et al.", "", t)

    # Remove Fig and Table
    t = re.sub("\( ?Fig [0-9]+ ?\)", "", t)
    t = re.sub("\( ?Table [0-9]+ ?\)", "", t)

    # Replace continuous spaces with a single space
    t = re.sub(' +', ' ', t)

    # Convert all to lowercase
    t = t.lower()
    return t

papers['abstract'] = papers['abstract'].astype(str).apply(clean_up)
papers['text'] = papers['text'].astype(str).apply(clean_up)
```

Here, we are using sklearn package to count the words frequency in each document.

```
from sklearn.feature_extraction.text import CountVectorizer
vectorizer = CountVectorizer()
# tokenize and build vocab
final_vector=vectorizer.fit_transform(papers['text'])
# summarize

vectorizer.vocabulary_

#TO KNOW FREQUENCY OF ONE WORD
vectorizer.vocabulary_.get(u'pulmonary')

print(final_vector.shape)
```

Disadvantage: The shape of final_vector is (40144,997130), which is a sparse matrix showing a big feature space.

Also it assumes all words are independent of each other, so doesn't find any correlation between words. Thirdly, Bag-of-Words gives equal importance to all the words, so those words which are not so much important have high frequency.
Because of these disadvantages, we switched to TF-IDF model which gives importance to rare words.

Snowball stemming algorithm

Snowball stemming algorithm is then applied on the dataframe. In natural language processing, stemming refers to the heuistic process in which certain rules are applied on a word to determine which characters to remove from the end of the word. The word then gets converted to a word stem, (which might not be the same as the original root of the word as in the dictionary, rather an equal or smaller form of the same word). Snowball stemmer (also called as Porter2 stemmer) is a widely accepted stemmer known for its handling of over-stemming, under-stemming and accuracy issues that other stemming algorithms suffer from.

```python
analyzer = CountVectorizer().build_analyzer()
stemmer = SnowballStemmer("english")

def preprocess(doc):
    doc=doc.lower()
    return str.join(" ", [stemmer.stem(w) for w in analyzer(doc)])

def preprocess_row(row):

    text = str.join(' ', [str(row.title), str(row.abstract), str(row.text)])
    return preprocess(text)

%time df['preprocessed'] = df.apply(lambda x: preprocess_row(x), axis=1)
```

The algorithm is that each row is first converted to a single lower case string on which Snowball stemming is applied. The stemmed data is then stored back into the dataframe.

For all the rows in the dataframe the following steps are followed:

1. Various column elements in the row are first converted to string format, then they are joined into a single string variable named text.
2. The variable text is then converted to lower case string.
3. This lower-cased string is then tokenised and Snowball stemming is applied on each token (here token means word, keyword etc.).
4. The stemmed tokens are rejoined into a single string and then this string is stored back into the dataframe.

Time statistics are also printed on the screen for the whole stemming operation on the dataframe.

The dataframe is now ready for the next step of TF-IDF application.

Applying TF-IDF

In a simple language, TF-IDF can be defined as follows:\
A High weight in TF-IDF is reached by a high term frequency(in the given document) and a low document frequency of the term in the whole collection of documents.
TF-IDF algorithm is made of 2 algorithms multiplied together.

Term Frequency

Term frequency (TF) is how often a word appears in a document, divided by how many words there are.

$TF(t)=NumberOfTimesTermTAppearsInADocumentTotalNumberOfTermsInTheDocument$

Inverse Document Frequency

Term frequency is how common a word is, inverse document frequency (IDF) is how unique or rare a word is.

$$IDF(t)=\log_e(TotalNumberOfDocumentsNumberOfDocumentsWithTermTInIt)$$

```
cv = CountVectorizer(max_df=0.95, stop_words='english')
word_count = cv.fit_transform(preprocessed)
tfidf_tr = TfidfTransformer(smooth_idf=True, use_idf=True)
tfidf_tr.fit(word_count)

def get_word_vector(document):
    w_vector = tfidf_tr.transform(cv.transform([document]))
    return w_vector
df['word_vector'] = df.preprocessed.apply(get_word_vector)
```

Using skLearn libraries we convert input document to TF-IDF matrix for further manipulation. In TF-IDF matrix each row will be treated as a vector for document. We will also form a TF-IDF vector of the query and then find the distance between them which will denote relevance.

How TF-IDF approach increases efficiency

BOW model depends on only TF frequencies i.e. the count of terms in documents which has many drawbacks 👎

1. Words which are present in most of the documents and are not stop words are not ignored and rareness of word is not considered.
2. BOW model builds a colossal vocabulary which is largely useless.

3. BOW model also creates heavily sparse vectors.

Calculate distance between searching phrase and each document

Firstly, we are calculating key words for each query sentence. Then we are calculating distance between each document and the searching document(phrase) using only words that were in query document(phrase)

What is Document Distance?

Two documents containing a huge amount of text. to know how similar these documents are, we will see how many words overlap in these documents.
Algorithm:

1. Open and read both documents. Only read words and numbers, skip special characters (spaces, dots, etc…) and convert the words to lower case.
2. Calculate the word frequency in both collections of words, this means how many times each word occur in each document.
3. Compare the frequencies from both computations and calculate the distance.

```
feature_names = cv.get_feature_names()
def get_words_with_value(w_vector):
    return sorted([(feature_names[ind], val) for ind, val in zip(w_vector.in
```

```python
def calculate_distance_between_words_vectors(search_words_indices, search_ve
    document_vec = document_vector[0, search_words_indices].toarray()
    return distance.euclidean(search_vec, document_vec)

def get_related_documents(text, number_of_documents):
    search_vector = get_word_vector(preprocess(text))
    search_words_indices = search_vector.indices
    search_vec = search_vector[0, search_words_indices].toarray()
    distance_idx = df.apply(lambda x: calculate_distance_between_words_vecto
    relevant_indexes = distance_idx.sort_values().head(number_of_documents).
    result_columns = ["title", "authors", "word_vector"]
    result = df[result_columns].iloc[relevant_indexes].fillna("")
    return result
```

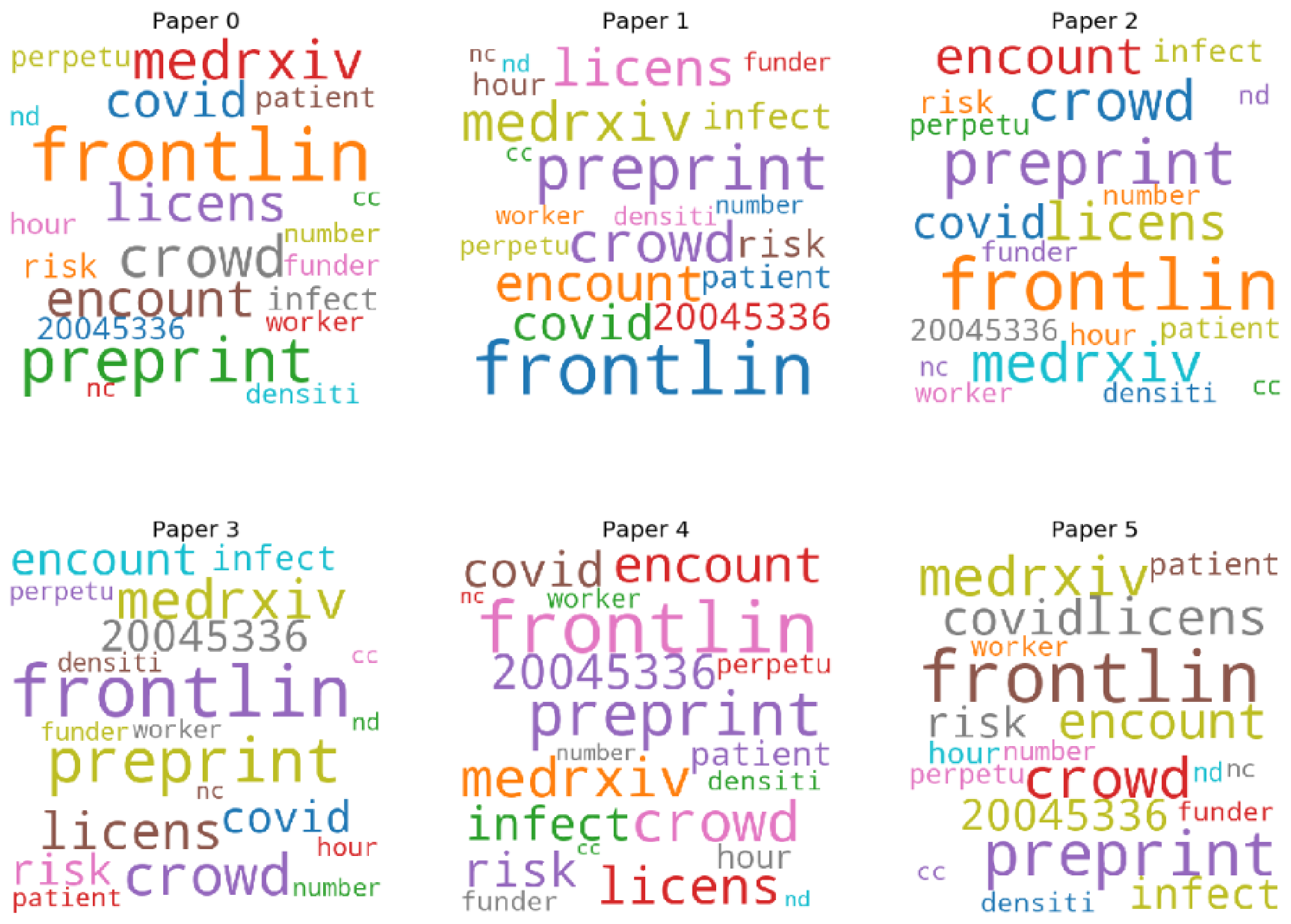
Two functions above: The first one, is calculating the distance between words vector and Second one is getting related documents once we have find the distance between them.

Similarity between documents is also tried out by finding cosine similarity and this approach is taken further. Unfortunately, we are getting different results from the euclidean approach. Cosine similarity is generally used as a metric for measuring distance when the magnitude of vectors doesn’t matter, hence it is perfect for working with text data. Because of all this, I prefer to find distance using Euclidean Approach and hence implemented it. Other Approach we can think of finding document distance is Manhattan distance, but it is L1 distance, so we skipped it and continued with L2 distance.

Results

Transmission dynamics of the virus, including the basic reproductive number, incubation period, serial interval, modes of transmission and environmental factors

	title	authors
0	Comparative Pathogenesis of Three Human and Zo...	B Rockx, F Feldmann, D Brining, D Gardner, R L...
1	A COVID-19 Infection Risk Model for Frontline ...	Louie Florendo Dy, Jomar Fajardo Rabajante
2	Deep sequencing reveals persistence of cell-as...	Sofia Morfopoulou, · Edward T Mee, Sarah M Con...
3	Strongly heterogeneous transmission of COVID-1...	Yuke Wang, Peter Teunis
4	Innate Immune Responses to Avian Influenza Vir...	Danyel Evseev, Katharine E Magor
5	Severe atypical pneumonia in critically ill pa...	S Valade, L Biard, V Lemiale, L Argaud, F Pène...


Paper 0
perpetu medrxiv covid patient nd frontlin licens cc hour number risk crowd funder encount infect 20045336 worker preprint nc densiti
Paper 1
nc nd licens funder hour medrxiv infect cc preprint worker densiti number perpetu crowd risk encount patient covid 20045336 frontlin
Paper 2
encount infect risk crowd nd perpetu preprint number covid licens funder frontlin 20045336 hour patient nc medrxiv worker densiti cc
Paper 3
encount infect perpetu medrxiv 20045336 densiti cc frontlin nd funder worker preprint nc licens covid hour risk crowd number patient
Paper 4
covid encount nc worker frontlin 20045336 perpetu preprint number patient medrxiv densiti infect crowd cc hour risk licens funder nd
Paper 5
medrxiv patient covid licens worker frontlin risk encount hour number perpetu crowd nd nc 20045336 funder cc preprint densiti infect