FIT5196 Task 2 in Assessment 1

Student Name: Anh Huy Phung

Student ID: 34140298

Student Name: Wei Yu Su

Student ID: 33642605

Date: 19/04/2024

Environment: Python 3.10

Libraries used:

- re (for regular expression, installed and imported)
- pandas (for dataframe, installed and imported)
- langdetect (for detecting the language of the text)
- itertools (for performing operations on iterables)
- nltk (Natural Language Toolkit, installed and imported)
- nltk.tokenize (for tokenization, installed and imported)
- nltk.stem (for stemming the tokens, installed and imported)
- nltk.probability (for representing and processing probabilistic information)
- nltk.util (for generating ngrams)
- nltk.corpus (for reuters)

Table of Contents

1. Introduction

In this task, we are going to write Python code to convert YouTube comments(in an Excel file) into numerical representations. This involved text extraction, tokenization, emojis and stopwords removal and other processes.

2. Importing Libraries

In this assessment, any python packages is permitted to be used. The following packages were used to accomplish the related tasks:

- **re** (for regular expression, installed and imported)
- **pandas** (for dataframe, installed and imported)
- **langdetect** (for detecting the language of the text)
- **itertools** (for performing operations on iterables)
- **nltk** (Natural Language Toolkit, installed and imported)
- **nltk.tokenize** (for tokenization, installed and imported)
- **nltk.stem** (for stemming the tokens, installed and imported)
- **nltk.probability** (for representing and processing probabilistic information)
- **nltk.util** (for generating ngrams)
- **nltk.corpus** (for reuters)

```
from google.colab import drive
drive.mount('/content/drive')
```

```
!pip install -q langdetect
!pip install -q pandarallel
```

```
import pandas as pd
import re
import multiprocessing as mp
from langdetect import detect
from langdetect import detect_langs
from langdetect import DetectorFactory
from langdetect.lang_detect_exception import LangDetectException
DetectorFactory.seed = 0
from pandarallel import pandarallel
from itertools import chain
from nltk.tokenize import RegexpTokenizer
from nltk.tokenize import MWETokenizer
from nltk.stem import PorterStemmer
import nltk
from nltk.probability import *
from nltk.util import ngrams
from nltk.corpus import reuters
from collections import Counter
```

Step 1: Data import

In this section, we are going to load the file (Group021).

```
def read_data(file_name):
    try:
        excel_data = pd.ExcelFile(file_name)
    except:
        print("Something went wrong when reading the file")
    finally:
        return excel_data
```

```
# Colab use this one: '/content/drive/Shareddrives/FIT5196_S1_2024/A1/Students data/Task 2/Group021.xlsx'
filename = "/content/drive/Shareddrives/FIT5196_S1_2024/A1/Students data/Task 2/Group021.xlsx"
excel_data = read_data(filename)
```

```
# Show all worksheets in this EXCEL file
excel_data.sheet_names
```

And let's parse and see the data in one of worksheet.

- `excel_data.parse(0)`

We can see the data is not positioned from the first column and it exists some `Unnamed` columns and `NaN` column values.

```
excel_data.parse(0)
```

Let's parse the data after loading, and also cleaning data in this step.

- We'll parse each worksheet data
- Use `dropna()` function to remove all rows and columns with `NULL` values
- Rename columns
- `Concat()` function to merge two DataFrame

```
def data_preprocess(excel_data):
    total_df = pd.DataFrame()
    log=[]
    try:
        for sheet_idx in excel_data.sheet_names:
            df = excel_data.parse(sheet_idx)
            df = df.dropna(axis=0, how = 'all')
            df = df.dropna(axis=1, how = 'all')
            unnamed_columns = [col for col in df.columns if 'Unnamed' in col]

            # if contains columns'Unnamed', then rename
            if unnamed_columns:
                df = df.drop(df.index[:1])
                df.columns = list(['id','snippet'])
                df.index = range(len(df.index))
            else:
                df.columns = list(['id','snippet'])

            total_df = pd.concat([total_df, df])
        total_df.drop_duplicates(inplace=True)
        total_df.index = range(len(total_df.index))
    except x:
        log.append(str(x))
        pass
    finally:
        return total_df
```

```
# data parsing and cleaning, remove nulls and set columns name
total_df = data_preprocess(excel_data)
```

```
total_df.head(5)
```

After pre-processing, we have

- 2 columns: `id, snippet`
- and **81824** rows

```
total_df.shape
```

Step 2: Text extraction and cleaning

In this step, we are going to..

1. Extract 'textOriginal' field
2. Remove emojis and normalise the text
3. Language detecting

2.1. Extract the 'textOriginal' field

In this step, 'textOriginal' fields required to extract in all top level comments.

- Extract channel_id
- Extract textOriginal

This function is to convert 'snippet' columns values into `json` type, then return our target text.

```
#str to json and catch target message
def json_txt_catch(row, is_channelID=False):
    """
    Convert a string representation of a JSON object stored in a DataFrame row into a Python dictionary.
    Extracts specific information from the dictionary, such as the channel ID and the original text message.

    Returns:
    - The function has an optional parameter 'is_channelID' which, if set to True, returns only the channel ID informat
    """
    log = []
    try:
        final_dictionary = eval(row['snippet'])
        textOriginal = [final_dictionary['channelId'], final_dictionary['topLevelComment']['snippet']['textOriginal']]
    except Exception as e:
        log.append(str(e))
        textOriginal = None
    finally:
        if is_channelID:
            return textOriginal[0]
        else:
            return textOriginal[1]
```

2.2. Remove emojis

One of the tasks is to remove emojis from text and normalise the text.

- To remove emojis, make sure your text data is in utf-8 format.
- The list of emojis to remove are in emoji.txt.

```
def load_emoji_txt(filename):
    """
    Load emoji from a file with encoding utf-8

    Returns:
    - emojiwords in a list
    """
    emojiwords = []
    try:
        with open(filename, encoding="utf-8") as f:
            emojiwords = f.read().splitlines()
    except:
        print('The file is not found')
    finally:
        return emojiwords
```

```
# Import emoji_txt file
emoji_file = "/content/drive/Shareddrives/FIT5196_S1_2024/A1/emoji.txt"
emojiwords = load_emoji_txt(emoji_file)
```

The above function, `load_emoji_txt()` function open the emoji_txt file with open() function, and return a file object.

```
def remove_emoji(string, emojiwords):
    """
    Removes emojis from text.

    Args:
    - text: The text from which emojis will be removed.

    Returns:
    - Text with emojis removed.
    """
    try:
        emoji_pattern = re.compile('|'.join(re.escape(emoji) for emoji in emojiwords))
        tem_txt = emoji_pattern.sub(r'', string)
    except:
        print('Something went wrong when removing emoji')
    finally:
        return tem_txt
```

```
str_emojis = "This is a string with emojis.🥇🥈🥉🆎🏧🅰😝"
print("The result: ", remove_emoji(str_emojis, emojiwords))
```

The above function, `remove_emoji()` function accpet two parameters.

1. `string`
2. `emojiwords`

The emojis list may exists illegal characters, we use `escape()` , then join each emojis with '|' , return a `string`.

At this stage, we remove the emojis and normalise the text into lower case by `lower()` function

---

2.3. Language detect

This step, we extract comments and detect language.

- Detect if the comment is English , If English then return `True`.

However, language detection sometimes is not accurate, we call the following code to enforce consistent results at the beginning.

`DetectorFactory.seed = 0`

```
def lang_dect(string):
    """
    Checks if text is in English.

    Args:
    - text: The text to be checked.

    Returns:
    - True if text is in English, False otherwise.
    """
    try:
        if not string:
            return False
        else:
            return detect(string) == 'en'
    except:
        return False
```

```
str_1 = "This is an English comment you are looking for."
str_2 = "おはようございます"
print("The result of str_1: ", lang_dect(str_1))
print("The result of str_2: ", lang_dect(str_2))
```

In **Step 2**, we have..

1. `json_txt_catch()`
2. `load_emoji_txt()`
3. `remove_emoji()`
4. `lang_dect()`

Before we generate a CSV file, we have to tidy up the data and process it in the format required, in which we will extract information (ChannelID & textOriginal) amd then remove it before defining all the english comments.

Regarding to filter our english comments, we will use pandarallel library to fastern the process and fill `fillna()` Null values with `" "`, because some comment only had emojis no text.

```
# Extracting ChannelID and textOriginal
total_df['ChannelID'] = total_df.apply(json_txt_catch, axis=1, is_channelID= True)
total_df['textOriginal'] = total_df.apply(json_txt_catch, axis=1, is_channelID= False)

# Remove emoji and lower string from textOriginal
total_df['textOriginal'] = total_df['textOriginal'].apply(lambda x: remove_emoji(x,emojiwords))
total_df['textOriginal'] = total_df['textOriginal'].apply(lambda x: x.lower())
```

```
total_df
```

```
# Use pandarallel to fastern the process
pandarallel.initialize(progress_bar=True, nb_workers= 4)
```

```
total_df['is_english'] = total_df['textOriginal'].parallel_apply(lang_dect)
```

```
# fill NA valud with space ' ' because there some empty comments after removing emoji
total_df.fillna(' ', inplace=True)
```

```
def channel_summary(df):
    """
    Generates a summary DataFrame containing aggregated information about channels.

    Args:
    - df: DataFrame containing channel data.

    Returns:
    - DataFrame with channel summary information including total comments and English comments count.
    """
    # Create a copy of the DataFrame to avoid modifying the original
    df_copy = df.copy()

    # Replace null values in 'textOriginal' with 1
    df_copy['textOriginal'].fillna(1, inplace=True)

    # Perform aggregation
    summary_df = df_copy.groupby('ChannelID').agg(
        total_comments=('textOriginal', 'count'),  # Count non-null values
        en_comments=('is_english', lambda x: (x == True).sum())
    ).reset_index()

    # Rename columns
    summary_df.columns = ['channel_id', 'all_comment_count', 'eng_comment_count']

    return summary_df
```

The above codes, we prepare the data before export to a csv file.

1. Create a copy of the DataFrame to avoid modifying the original
2. Replace null values in 'textOriginal' with 1
3. Perform aggregation
4. Rename columns

```
def english_df(summary_df, df):
    """
    Filters DataFrame to include only English comments from channels with at least 15 English comments.

    Args:
    - summary_df: Summary DataFrame containing channel information.
    - df: Original DataFrame containing comments.

    Returns:
    - Filtered DataFrame containing English comments from selected channels.
    """
    # Filter channels with at least 15 English comments
    filter_id = summary_df['channel_id'][summary_df['eng_comment_count'] >= 15]

    # Filter DataFrame to include only selected channels and English comments
    filtered_channels = df[df['ChannelID'].isin(filter_id)]
    filtered_channels = filtered_channels[filtered_channels['is_english'] == True]

    return filtered_channels[['ChannelID', 'textOriginal']]
```

```
summary_df = channel_summary(total_df)
eng_df = english_df(summary_df, total_df)
eng_df
```

	ChannelID	textOriginal
0	UCsT0YIqwnpJCM-mx7-gSA4Q	at 215 pounds (i was 140 pounds in high school...
1	UCNye-wNBqNL5ZzHSJj3l8Bg	too much population
2	UCNye-wNBqNL5ZzHSJj3l8Bg	dr prof. valentina zharkova\r\n\r\nso the eart...
3	UCNye-wNBqNL5ZzHSJj3l8Bg	democratic elections have a very large carbon ...
6	UCBJycsmduvYEL83R_U4JriQ	2nd most popular video in 2 weeks
...	...	...
81814	UCX6b17PVsYBQ0ip5gyeme-Q	um....is anybody even out here in this comment...
81815	UCZAc6j2gnLgbVtvZ9F_biQg	this looks so fun to make! i want to make these!
81816	UCZAc6j2gnLgbVtvZ9F_biQg	may i know how many laddoos were made out of t...
81817	UCT-VzthVAM_4ohDdKa-BbXA	what is quantum computer? it's look like giant...
81818	UCqoAEDirJPjEUFcF2FklnBA	i feel like chuck is now most likely one of th...

50266 rows × 2 columns

```
eng_df.shape
```

Step 3: Generate CSV file

Generate a **csv** file that contains unique channel ids along with the counts of top level comments(all language, and english).

The column names are:

- channel_id
- all_comment_count
- eng_comment_count

Double-click (or enter) to edit

And then `to_csv()` to a csv file.

```
# Write the formatted summary ChannelID to a text file
summary_df.to_csv('/content/drive/MyDrive/021_channel_list.csv', index=False)
```

Step 4: Generate the unigram and bigram lists and output as vocab

In this step, we have serval processes to do, each process has been separated into small sections by function.

- **Tokenization**
- **Stopwords removal**
- **Porter stemmer**
- **Rare tokens removal**
- **Tokens with a lenth less than 3 removal**
- **Bigram using PMI scire**

The flow will go through:

1. **Tokenization**
2. **Bigram**
3. **Remove stopwords/ Rare tokens/ Less than 3 words removal**
4. **Stem tokens**
5. **Create final vocab**

We choose to extract bigrams after tokenization to prevent bias in the data (skewness) caused by removing unimportant words like stopwords, rare tokens, and tokens with less than three characters. This sequence ensures a more accurate representation of the text.

4.1. Tokenization

This function tokenizes the words in each comment using a regular expression and these list of tokens by aggregate by channelID level.

- regular expression pattern : `[a-zA-Z]+`

```
def regexp_tokenizer(text):
    """
    Tokenize text using regular expressions.

    Parameters:
        text (str): The input text.

    Returns:
        list: List of tokens.
    """
    tokenizer = RegexpTokenizer(r'[a-zA-Z]+')
    unigram_tokens = tokenizer.tokenize(text)
    return unigram_tokens


def tokens_in_channelID_df(df):
    """
    Aggregate words by ChannelID.

    Parameters:
        df (DataFrame): Input DataFrame containing text data.

    Returns:
        DataFrame: DataFrame with word lists aggregated by ChannelID.
    """
    df = df.copy()
    df['word_list'] = df['textOriginal'].apply(lambda x: regexp_tokenizer(x))
    tokemlist_by_channelID_df = df.groupby('ChannelID')['word_list'].agg(lambda x: sum(x, []))
    return tokemlist_by_channelID_df.reset_index()
```

```
# This is demonstration about tokenize all english comment and aggregate by ChannelID levels
token_df_test = tokens_in_channelID_df(eng_df)
token_df_test
```

At this stage, all comments for each Channel_ID are tokenized.

4.2. Bigram Generation

This function is designed to extract bigrams from a given list of tokens and then verify whether these bigrams occur together within comments. If both words of a bigram are found in at least one comment, it is considered a valid bigram; otherwise, it is removed from the list of bigrams.

1. **bigrams:** Extracts bigrams from a token lists using the PMI measure and returns the top 200 bigrams.
2. **contains_bigram:** Checks if any bigram from a given list is present in a text string and returns the list of bigrams found in the text.
3. **bigram_check:** Searches for specific bigrams within a DataFrame containing text data and returns a list of bigrams that exist in at least one comment in the DataFrame.

```
def bigrams(words_list):
    """
    Extract bigrams from the list of words.

    Parameters:
        words_list (list): List of input words.

    Returns:
        list: List of bigrams.
    """
    bigram_measures = nltk.collocations.BigramAssocMeasures()
    finder = nltk.collocations.BigramCollocationFinder.from_words(words_list)
    return finder.nbest(bigram_measures.pmi, 200)


def contains_bigram(text, bigrams):
    """
    Check if any word from the given list is present in the text.

    Parameters:
        text (str): The input text.
        word_list (list): List of words to search for.

    Returns:
        list: List of words found in the text.
    """
    bigrams_found = []
    for bigram in bigrams:
      if bigram[0] in text and bigram[1] in text:
        bigrams_found.append(bigram)
    return bigrams_found

def bigram_check(eng_df, bigrams):
    """
    Check for bigrams in the DataFrame.

    Parameters:
        eng_df (DataFrame): Input DataFrame containing text data.
        bigram (list): List of bigrams to search for.

    Returns:
        List of bigram that exist at least one comment
    """
    full_col_list = [' '.join(element) for element in bigrams]
    # eng_df['text_testing'] = eng_df['textOriginal'].apply(lambda x: contains_word(x, full_col_list))
    eng_df['text_testing'] = eng_df['textOriginal'].apply(lambda x: contains_bigram(x, bigrams))
    final_dict = dict(sum((Counter(d) for d in eng_df['text_testing']), Counter()))
    sorted_final_dict = dict(sorted(final_dict.items(), key=lambda item: item[1]))
    # print(sorted_final_dict)

    final_bigram_set = []
    for bigram in bigrams:
      if bigram in sorted_final_dict:
        final_bigram_set.append(bigram)
    return final_bigram_set
```

```
    # From above token list, we extract bigram and check it
    token_list_test = token_df_test['word_list'].tolist()
    token_list_test = list(chain.from_iterable(token_list_test))
    # print(len(token_list_test))
    bigrams_two_hundred= bigrams(token_list_test)
    final_bigram_set_test = bigram_check(eng_df,bigrams_two_hundred)

    bigrams_not_found = [bigram for bigram in bigrams_two_hundred if bigram not in final_bigram_set_test]

    print('200 bigrams: \n', bigrams_two_hundred)
    print('Chosen bigrams: \n', final_bigram_set_test)
    print('Remove bigrams: \n', bigrams_not_found)
```

4.2. Stopwords, rare tokens, less than 3 words tokens removal

This section decribes following steps in handling tokens, includes:

1. **stopwords_final_set:** Determines the final set of stopwords to be removed from the text data. It identifies context-dependent stopwords and then combines with a predefined set of independent stopwords to form the final stopwords set.
2. **rare_tokens_set:** Identifies rare tokens in the text data which occurs in less than 1% of the channels and returned as a set
3. **less_three_tokens_remove:** Removes tokens with fewer than three characters from a given list of tokens
4. **eliminate_or_choose_words:** This function either chooses or eliminates words from a given list based on a provided set of words. The indicator parameter determines whether words from the input set should be chosen (True) or eliminated (False).

```
    def load_stopwords_txt(filename):
        stopwords = []
        try:
            with open(filename) as f:
                stopwords = f.read().splitlines()
        except:
            print('The file is not found')
        finally:
            return stopwords
```

```
    #load stopwords_txt
    stopwords_file = "/content/drive/Shareddrives/FIT5196_S1_2024/A1/stopwords_en.txt"
    stop_word_list = load_stopwords_txt(stopwords_file)
```

The above code loads the stopwords_en file.

```
    def stopwords_final_set(df, stop_word_list):
        """
        Determine final set of stopwords.

        Parameters:
            df (DataFrame): Input DataFrame containing text data.

        Returns:
            set: Final set of stopwords, including both independent and context-dependent ones.
        """
        # Identify context-dependent stopwords in a dataframe with threshold is 0.99
        word_id_counts = df.explode('word_list').groupby('word_list')['ChannelID'].nunique()
        total_ids = df['ChannelID'].nunique()
        percentile_threshold = 0.99 * total_ids
        context_dependent_words = word_id_counts[word_id_counts > percentile_threshold].index.tolist()

        # Combine it with independent stopwords to have final stopwords set
        stopwords_final_set = set(context_dependent_words + stop_word_list)
        return stopwords_final_set

    def rare_tokens_set(df):
        """
        Identify rare tokens in a dataframe with threshold is 0.01.

        Parameters:
            df (DataFrame): Input DataFrame containing text data.

        Returns:
            set: Set of rare tokens.
        """
        word_id_counts = df.explode('word_list').groupby('word_list')['ChannelID'].nunique()
        total_ids = df['ChannelID'].nunique()
        percentile_threshold = 0.01 * total_ids
        rare_tokens = word_id_counts[word_id_counts < percentile_threshold].index.tolist()
        return set(rare_tokens)

    def less_three_tokens_remove(input_list):
        """
        Remove tokens with less than three characters.

        Parameters:
            input_list (list): List of input tokens.

        Returns:
            list: Filtered list of tokens.
        """
        return [element for element in input_list if len(element) >= 3]

    def eliminate_or_choose_words(word_list, input_set, indicator=True):
        """
        Choose or eliminate words based on input set.

        Parameters:
            word_list (list): List of input words.
            input_set (set): Set of words to choose or eliminate.
            indicator (bool): Indicator to choose or eliminate.

        Returns:
            list: Filtered list of words.
        """
        try:
            word_list = list(word_list)
            if indicator:
                word_list = [word for word in word_list if word not in input_set]
            else:
                word_list = [word for word in word_list if word in input_set]
        except Exception as e:
            print(f"Error occurred: {e}")
        return word_list
```

```
# Print all stopwords, including independent and dependent-context stopwords
stopwords_set = stopwords_final_set(token_df_test, stop_word_list)
print('All stopwords: \n', stopwords_set)

# Print rare tokens
rare_tokens = rare_tokens_set(token_df_test)
print('All rare tokens: \n', rare_tokens)

# Remove all stopwords in token lists
token_list_test = eliminate_or_choose_words(token_list_test, stopwords_set, True)

# Remove all rare tokens in token lists
token_list_test = eliminate_or_choose_words(token_list_test, rare_tokens, True)

# Remove all less then 3 token words in token lists
token_list_test = less_three_tokens_remove(token_list_test)
```

4.3. Stem tokens

Below also shows us how to stem tokens from the tokens list. Here we also apply **pool.map** from **pandarallel** library to enhace stem efficiency:

1. **word_stem:** Stems all the words from an input list using a PorterStemmer
2. **parallel_word_stem:** Divides the input list into smaller chunks and distributes them across multiple processes for stemming, thereby enhancing efficiency.

```
def word_stem(input_list):
    """
    Stem words in the input list.

    Parameters:
        input_list (list): List of input words.

    Returns:
        list: List of stemmed words.
    """
    stemmer = PorterStemmer()
    final_tokens_stemmed = [stemmer.stem(token) for token in input_list]
    return final_tokens_stemmed

def parallel_word_stem(input_list):
    """
    Stem words in the input list in parallel.

    Parameters:
        input_list (list): List of input words.

    Returns:
        list: List of stemmed words.
    """
    chunk_size = 1000
    with mp.Pool() as pool:
        results = pool.map(word_stem, [input_list[i:i+chunk_size] for i in range(0, len(input_list), chunk_size)])
    return [item for sublist in results for item in sublist]
```

```
# This is an example stems all the tokens in a list
stem_list_test = parallel_word_stem(token_list_test)
print(stem_list_test[:5])
```

4.5. Vocab output

This function serves to provide final vocabulary (both unigrams and bigrams) and related details required for subsequent processing steps. It orchestrates a sequence of operations, including tokenization, stop word removal, elimination of rare tokens, and removal of tokens with fewer than three characters. Additionally, it facilitates the extraction of bigrams at various stages of preprocessing.

```
"""
  This function delivers the vocabulary and related information necessary for further processing.

  Parameters:
      df (pd.DataFrame): DataFrame containing comment data.
      stop_word_list (list): List of stop words to be filtered.
      ordered_functions (list): List of functions to be applied in a specific order.
      bigram_step (str): Indicates at which step the bigram extraction should occur.

  Returns:
      tuple: Tuple containing information about bigrams, final bigram set, sorted total vocabulary, vocabulary index di
"""
def vocab_output(df, stop_word_list, ordered_functions, bigram_step):
    bigram, total_vocab_sorted, vocab_index_dic, stemmed_tokens_list, stemmed_unigram_sett = None, None, None, None, No
    for func in ordered_functions:
        # Tokenization
        if func.__name__ == 'tokens_in_channelID_df':
            tokens_channelID_df = func(df)
            token_list = tokens_channelID_df['word_list'].tolist()
            token_list = list(chain.from_iterable(token_list))
            # Extract bigrams and check it existence in comment
            if bigram_step == 'tokens_in_channelID_df':
                words = token_list.copy()
                bigram = bigrams(token_list)
                final_bigram_set = bigram_check(df, bigram)

        # Remove stopwords
        elif func.__name__ == 'eliminate_or_choose_words':
            stopwords_set = stopwords_final_set(tokens_channelID_df, stop_word_list)
            token_list = func(token_list, stopwords_set, True)
            # Extract bigrams and check it existence in comment
            if bigram_step == 'eliminate_or_choose_words':
                words = token_list.copy()
                bigram = bigrams(token_list)
                final_bigram_set = bigram_check(df, bigram)

        # Remove stopwords
        elif func.__name__ == 'rare_tokens_set':
            rare_tokens = func(tokens_channelID_df)
            token_list = [w for w in token_list if w not in rare_tokens]
            # Extract bigrams and check it existence in comment
            if bigram_step == 'rare_tokens_set':
                words = token_list.copy()
                bigram = bigrams(token_list)
                final_bigram_set = bigram_check(df, bigram)

        # Remove less then 3 words tokens
        elif func.__name__ == 'less_three_tokens_remove':
            token_list = less_three_tokens_remove(token_list)
            # Extract bigrams and check it existence in comment
            if bigram_step == 'less_three_tokens_remove':
                words = token_list.copy()
                bigram = bigrams(token_list)
                final_bigram_set = bigram_check(df, bigram)

    # Stem all tokens after being tokenizedd, remove stopwords/ rare tokens and less than 3 words tokens
    stemmed_unigram_list = parallel_word_stem(token_list)
    stemmed_unigram_set = set(stemmed_unigram_list)

    # Combine unigrams and bigrams and put it MWETokenizer, which helps us to tokenize comment level in following funct
    uni_voc = list(set(stemmed_unigram_list))
    for element in final_bigram_set:
        uni_voc.append(element)

    mwe_tokenizer = MWETokenizer(uni_voc)

    # Join bigram together with ungrams to generate final vocab
    total_vocab = ['_'.join(collocation) for collocation in final_bigram_set]
    total_vocab.extend(list(set(stemmed_unigram_list)))
    total_vocab_sorted = sorted(total_vocab)

    # Create dictionary contain token and its index
    vocab_index_dic = {word: i for i, word in enumerate(total_vocab_sorted)}
    return bigram, final_bigram_set, total_vocab_sorted, vocab_index_dic, words, mwe_tokenizer, stemmed_unigram_set
```

```
# Order of process function at is tokenization by comment level, remove stopwords, remove rare tokens and remove less t
functions = [tokens_in_channelID_df, eliminate_or_choose_words, rare_tokens_set, less_three_tokens_remove]

# Here extract all information and take bigrams at after tokenization whose bigram_step is 'tokens_in_channelID_df'
bigram, final_bigram_set, total_vocab_sorted, vocab_index_dic, words, mwe_tokenizer, stemmed_unigram_set = vocab_output
```

```
print(f'Total {len(final_bigram_set)} bigrams: \n', final_bigram_set)
print(f'Total {len(total_vocab_sorted)} vocab: \n', total_vocab_sorted)
print(f'Total {len(stemmed_unigram_set)} stemmed unigrams: \n', stemmed_unigram_set)
print('Indexed dictionary: \n',vocab_index_dic)
```

```
Total 195 bigrams:
 [('aalur', 'bharta'), ('aapl', 'msci'), ('abbabba', 'camboaddias'), ('abhinav', 'srivastava'), ('aboot', 'yoou'), ('abr
Total 3118 vocab:
 ['aalur_bharta', 'aapl_msci', 'abandon', 'abbabba_camboaddias', 'abhinav_srivastava', 'abil', 'aboot_yoou', 'abra_bulac
Total 2923 stemmed unigrams:
 {'korean', 'yeah', 'unknown', 'consciou', 'he', 'expens', 'quiet', 'commit', 'jazz', 'easili', 'umm', 'circl', 'instal
Indexed dictionary:
 {'aalur_bharta': 0, 'aapl_msci': 1, 'abandon': 2, 'abbabba_camboaddias': 3, 'abhinav_srivastava': 4, 'abil': 5, 'aboot_
```

Step 5: Numerical representation and output

Before going directly to generate numerical representation, we introduce some of below functions to help us simplify some steps require in the assignment specification:

1. **bigram_in_comment:** Return all the bigrams from an input list
2. **freq_index_dic:** Generates a dictionary containing the index and frequency of each token based on the provided frequency distribution and vocabulary index dictionary.
3. **convert_dic:** Generates text concatenating all values in a dictionary

```
def bigram_in_comment(input_list):
    """
    Filters strings in the input list to include only those containing an underscore character and having exactly two c

    Parameters:
        input_list (list): A list of strings.

    Returns:
        list: A new list containing only strings with an underscore and exactly two components.
    """
    return [string for string in input_list if '_' in string and len(string.split('_')) == 2]

def freq_index_dic(freq_dist, vocab_index_dic):
    """
    Generates a dictionary containing the index and frequency of each token based on the provided frequency distributio

    Parameters:
        freq_dist (nltk.probability.FreqDist): Frequency distribution of tokens.
        vocab_index_dic (dict): Dictionary mapping tokens to their corresponding indices.

    Returns:
        dict: Dictionary containing token index-frequency pairs.
    """
    token_info = {vocab_index_dic[token]: frequency for token, frequency in freq_dist.items()}
    return token_info

def convert_dic(input_dic, line_break=False):
    """
    Generates text concatenating all values in a dictionary, with the option to include line breaks between key-value p

    Parameters:
        input_dic (dict): Dictionary containing key-value pairs.
        line_break (bool): Flag indicating whether to include line breaks between key-value pairs.

    Returns:
        str: Concatenated text representation of the dictionary.
          line_break (False): Create format for key and value with key:value and join it with comma (,)
          line_break (True): Create format for key and value with key,value and join it line_break (\n)
    """
    try:
      if line_break is False:
          result_string = ','.join([f"{key}:{value}" for key, value in input_dic.items()])
      else:
          result_string = '\n'.join([f"{key},{value}" for key, value in input_dic.items()])
    except Exception as e:
      print(f"Error': {e}")
    return result_string
```

```
# Get FreqDist of all stemmed unigram tokens
freq_dist = FreqDist(stem_list_test)

# Test bigram_in_comment function, get all bigrams from a sample list, in this case is total_vocab_sorted
print("Testing bigram_in_comment:")
print(bigram_in_comment(total_vocab_sorted), '\n')

# Test freq_index_dic function, return all token's index and token's frequency in a dictionfreq_index_dicndex_dicfreq_i
freq_index_dict = freq_index_dic(freq_dist, vocab_index_dic)
print("Testing freq_index_dict:", freq_index_dict, '\n')

# Test convert_dic function, return all token's index and token's frequency (format: key,value) and then join all with
print("Testing convert_dic:", convert_dic(freq_index_dict, line_break=False), '\n')
```

The primary idea behind generating numerical representations involves tokenizing English comments using the MWETokenizer, filtering out vocabulary for each ChannelID, and outputting the sparse numerical representation. Here's a description of the functions below:

1. **Tokenization with MWETokenizer:** Utilizes the MWETokenizer obtained in the vocab_output() function to tokenize all comments in the eng_df dataFrame.
2. **Filtering Bigrams and Stemming Tokens:** Filters out all bigrams using bigram_in_comment() and places them into a separate column named 'collocation'. Then, stems all tokens and selects only unigrams in the vocabulary and put it into column name 'tokens_list'. Reason to filter collocation out first because we want to avoid accidently stem the collocaiton and it will not match the orginal vocab.
3. **Combining Bigrams and Unigrams with FreqDist():** Combines all bigrams and unigrams for each ChannelID using the FreqDist() function, then represents tokens with their respective indices.
4. **Generating Sparse Numerical Representation:** Finally, returns a dictionary where each key is a ChannelID mapped to its sparse numerical representation.

```
def countvec_output(eng_df, stemmed_unigram_set, vocab_index_dic, mwe_tokenizer, total_vocab_sorted):
    """
    Generates a dictionary containing the index and frequency of each token for each ChannelID in the provided DataFran

    Parameters:
        eng_df (pd.DataFrame): DataFrame containing comment data.
        stop_word_list (list): List of stop words to be filtered.
        stemmed_unigram_set (set): Set of stemmed tokens.
        vocab_index_dic (dict): Dictionary mapping tokens to their corresponding indices.
        mwe_tokenizer (Tokenizer): Multi-word expression tokenizer.
        total_vocab_sorted (list): List of total vocabulary sorted in descending order.

    Returns:
        A dictionary of ChannelID and its sparse representation pairs.
    """
    # Step 1: Aggregate all tokens by ChannelID level
    tokens_channelID_df = tokens_in_channelID_df(eng_df)

    # Step 2: Tokenize each comment using mwe_tokenizer
    tokens_channelID_df['tokens_list'] = tokens_channelID_df['word_list'].apply(lambda x: mwe_tokenizer.tokenize(x))

    # Step 3: Handle collocations (bigrams) in the comments
    tokens_channelID_df['collocation'] = tokens_channelID_df['tokens_list'].apply(lambda x: bigram_in_comment(list(x))
    tokens_channelID_df['collocation'] = tokens_channelID_df['collocation'].apply(lambda x: eliminate_or_choose_words(

    # Step 4: Stem all the unigrams and filter out ones in the unigram vocab only
    tokens_channelID_df['tokens_list'] = tokens_channelID_df['tokens_list'].apply(lambda x: parallel_word_stem(list(x))
    tokens_channelID_df['tokens_list'] = tokens_channelID_df['tokens_list'].apply(lambda x: eliminate_or_choose_words(
    # Step 5: Combine total vocabulary of each comment in a column 'freq_word_list'
    tokens_channelID_df['freq_word_list'] = tokens_channelID_df['tokens_list'] + tokens_channelID_df['collocation']

    # Step 6: Convert the 'freq_word_list' column to frequency distributions
    tokens_channelID_df['freq_word_list'] = tokens_channelID_df['freq_word_list'].apply(lambda x: FreqDist(list(x)))

    # Step 7: Create sparse representation using freq_index_dic and convert_dic functions
    tokens_channelID_df['freq_word_list'] = tokens_channelID_df['freq_word_list'].apply(lambda x: freq_index_dic(x, voc
    tokens_channelID_df['freq_word_list'] = tokens_channelID_df['freq_word_list'].apply(lambda x: convert_dic(x, False)

    # Step 8: Create a dictionary containing each ChannelID as key and its sparse representation as value
    channel_dic = dict(zip(tokens_channelID_df['ChannelID'], tokens_channelID_df['freq_word_list']))

    return channel_dic
```

```
# A dictionary of ChannelID and its sparse representation pairs
final_dic = countvec_output(eng_df, stemmed_unigram_set, vocab_index_dic, mwe_tokenizer, total_vocab_sorted)
```

Output files

```
# Write the formatted _countvec to a text file
with open('/content/drive/MyDrive/021_countvec.txt', 'w') as f:
    result_string_linebreak = convert_dic(final_dic, True)
    f.write(result_string_linebreak)

# Write the formatted _vocab to a text file
with open('/content/drive/MyDrive/021_vocab.txt', 'w') as f:
    for key, value in vocab_index_dic.items():
      f.write(f"{key}:{value}\n")
```

```
import pandas as pd
import itertools

groupnum = input("Please input your group number:")
df = pd.read_csv("/content/drive/MyDrive/{}_channel_list.csv".format(groupnum.zfill(3)))
df_col = ["channel_id", "all_comment_count", "eng_comment_count"]
assert all(df.columns == df_col) == True, "check your csv columns!"

print("Task 2 csv file passed!")

with open("/content/drive/MyDrive/{}_vocab.txt".format(groupnum.zfill(3)), "r") as file:
    vocab = file.readlines()
try:
    vocab = [each.strip().split(":") for each in vocab]
except:
    raise ValueError("Vocab file structured incorrectly!")

print("Task 2 vocab file passed!")

with open("/content/drive/MyDrive/{}_countvec.txt".format(groupnum.zfill(3)), "r") as file:
    countvec = file.readlines()

countvec = [each.strip().split(",") for each in countvec]
assert (
    all([":" not in each[0] for each in countvec]) == True
), "The channel id in countvec doesn't look right!"
try:
    allcounts = list(itertools.chain.from_iterable([each[1:] for each in countvec]))
    ind_counts = [each.split(":") for each in allcounts]
    # testing whether the ind:count can be parsed as numerical values
    [(int(each[0]), int(each[1])) for each in ind_counts]
except:
    raise ValueError("The ind:count part of your countvec doesnt look right!")

print("Task 2 countvec file passed!")
```

6. Summary

Our approach to task 2 shows:

1. Import the `xlsx` file by `pd.ExcelFile` and `concat()` all worksheets. Remove all `NULL` values and rename columns.
2. Extract **'textOriginal'** from YouTube snippets, then **remove emojis** and **language detection** and count the comments for the CSV file required.
3. Then, in the next step, the comments are **Tokenized** and converted into **Bigram**, **stopwords/ Rare tokens/ the tokens with a length of less than 3 words** are removed from the tokens list
4. Then **Stems** all the words from an input list using a PorterStemmer, by multiple processes for stemming to enhance efficiency.
5. A sequence of operations, including **tokenization**, **stop word removal**, **elimination of rare tokens**, and **removal of tokens with fewer than three characters.**
6. **Bigrams** from an input list, then generates a dictionary containing the index and frequency of each token, and generates a dictionary concatenated all values.
7. Create **sparse representation** for each ChannelID with desirable format.

---

7. References

[1] Pandas dataframe.drop_duplicates(), https://www.geeksforgeeks.org/python-pandas-dataframe-drop_duplicates/, Accessed 13/08/2022.

----

8. Workspace link

Link to my workspace