## Setup

- Code to download the data directly from the colab notebook.
- If you find it easier to download the data from the kaggle website (and uploading it to your drive), you can skip this section.

## Copy this notebook (if using Colab) via `File -> Save a Copy in Drive`.

You can do this assignment outside of Colab (using your local Python installation) via `File -> Download`.

**Use the "Text" blocks to provide explanations wherever you find them necessary. Highlight your answers inside these text fields to ensure that we don't miss it while grading your HW.**

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

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
```

```
# First mount your drive before running these cells.
# Create a folder for the this HW and change to that dir
%cd drive/MyDrive/CSE519
```

```
/content/drive/MyDrive/CSE519
```

## Download data from Kaggle

```
!pip install -q kaggle
```

```
from google.colab import files
# Create a new API token under "Account" in the kaggle webpage and download the json file
# Upload the file by clicking on the browse
files.upload()
```

Choose Files No file chosen Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.

```
Saving kaggle.json to kaggle.json
{'kaggle.json': b'{"username":"philipanish011","key":"f284f1b88c53e1e7c687a0f16089e902"}'}
```

```
! mkdir ~/.kaggle
```

```
! cp kaggle.json ~/.kaggle/
```

```
!kaggle competitions download -c commonlit-evaluate-student-summaries
```

```
commonlit-evaluate-student-summaries.zip: Skipping, found more recently modified local copy (use --force to force download)
```

## Alternate: download data using gdown (if having issues with Kaggle)

```
!pip install gdown
```

```
import gdown
url = 'https://drive.google.com/uc?id=164sQHZYvxU2XXPokrjzqv9MCGAMHaCIM'
gdown.download(url)
```

```
Downloading...
From: https://drive.google.com/uc?id=164sQHZYvxU2XXPokrjzqv9MCGAMHaCIM
To: /content/commonlit-evaluate-student-summaries.zip
100%|██████████| 1.10M/1.10M [00:00<00:00, 29.6MB/s]
'commonlit-evaluate-student-summaries.zip'
```

# Extract data and install packages (regardless of data acquisition method)

```
!unzip commonlit-evaluate-student-summaries.zip
```

```
Archive:  commonlit-evaluate-student-summaries.zip
  inflating: prompts_test.csv
  inflating: prompts_train.csv
  inflating: sample_submission.csv
  inflating: summaries_test.csv
  inflating: summaries_train.csv
```

```
### TODO: Install required packages
### Student's code here
!pip install pandas
!pip install scikit-learn
!pip install numpy
!pip install matplotlib
!pip install seaborn
!pip install profanity
!pip install import-java
!pip install language-tool-python

### END
```

```
Requirement already satisfied: pandas in /usr/local/lib/python3.10/dist-packages (1.5.3)
Requirement already satisfied: python-dateutil>=2.8.1 in /usr/local/lib/python3.10/dist-packages (from pandas) (2.8.2)
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-packages (from pandas) (2023.3.post1)
Requirement already satisfied: numpy>=1.21.0 in /usr/local/lib/python3.10/dist-packages (from pandas) (1.23.5)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-packages (from python-dateutil>=2.8.1->pandas) (1.16.0)
Requirement already satisfied: scikit-learn in /usr/local/lib/python3.10/dist-packages (1.2.2)
Requirement already satisfied: numpy>=1.17.3 in /usr/local/lib/python3.10/dist-packages (from scikit-learn) (1.23.5)
Requirement already satisfied: scipy>=1.3.2 in /usr/local/lib/python3.10/dist-packages (from scikit-learn) (1.11.2)
Requirement already satisfied: joblib>=1.1.1 in /usr/local/lib/python3.10/dist-packages (from scikit-learn) (1.3.2)
Requirement already satisfied: threadpoolctl>=2.0.0 in /usr/local/lib/python3.10/dist-packages (from scikit-learn) (3.2.0)
Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-packages (1.23.5)
Requirement already satisfied: matplotlib in /usr/local/lib/python3.10/dist-packages (3.7.1)
Requirement already satisfied: contourpy>=1.0.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib) (1.1.0)
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.10/dist-packages (from matplotlib) (0.11.0)
Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3.10/dist-packages (from matplotlib) (4.42.1)
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib) (1.4.5)
Requirement already satisfied: numpy>=1.20 in /usr/local/lib/python3.10/dist-packages (from matplotlib) (1.23.5)
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.10/dist-packages (from matplotlib) (23.1)
Requirement already satisfied: pillow>=6.2.0 in /usr/local/lib/python3.10/dist-packages (from matplotlib) (9.4.0)
Requirement already satisfied: pyparsing>=2.3.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib) (3.1.1)
Requirement already satisfied: python-dateutil>=2.7 in /usr/local/lib/python3.10/dist-packages (from matplotlib) (2.8.2)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-packages (from python-dateutil>=2.7->matplotlib) (1.16.0)
Requirement already satisfied: seaborn in /usr/local/lib/python3.10/dist-packages (0.12.2)
Requirement already satisfied: numpy!=1.24.0,>=1.17 in /usr/local/lib/python3.10/dist-packages (from seaborn) (1.23.5)
Requirement already satisfied: pandas>=0.25 in /usr/local/lib/python3.10/dist-packages (from seaborn) (1.5.3)
Requirement already satisfied: matplotlib!=3.6.1,>=3.1 in /usr/local/lib/python3.10/dist-packages (from seaborn) (3.7.1)
Requirement already satisfied: contourpy>=1.0.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib!=3.6.1,>=3.1->seaborn) (1.1
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.10/dist-packages (from matplotlib!=3.6.1,>=3.1->seaborn) (0.11.0)
Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3.10/dist-packages (from matplotlib!=3.6.1,>=3.1->seaborn) (4.
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib!=3.6.1,>=3.1->seaborn) (1.
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.10/dist-packages (from matplotlib!=3.6.1,>=3.1->seaborn) (23.1
Requirement already satisfied: pillow>=6.2.0 in /usr/local/lib/python3.10/dist-packages (from matplotlib!=3.6.1,>=3.1->seaborn) (9.4.0)
Requirement already satisfied: pyparsing>=2.3.1 in /usr/local/lib/python3.10/dist-packages (from matplotlib!=3.6.1,>=3.1->seaborn) (3.1
Requirement already satisfied: python-dateutil>=2.7 in /usr/local/lib/python3.10/dist-packages (from matplotlib!=3.6.1,>=3.1->seaborn)
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-packages (from pandas>=0.25->seaborn) (2023.3.post1)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-packages (from python-dateutil>=2.7->matplotlib!=3.6.1,>=3.1-
Requirement already satisfied: profanity in /usr/local/lib/python3.10/dist-packages (1.1)
Requirement already satisfied: import-java in /usr/local/lib/python3.10/dist-packages (0.6)
Requirement already satisfied: pyjnius in /usr/local/lib/python3.10/dist-packages (from import-java) (1.5.0)
Requirement already satisfied: language-tool-python in /usr/local/lib/python3.10/dist-packages (2.7.1)
Requirement already satisfied: requests in /usr/local/lib/python3.10/dist-packages (from language-tool-python) (2.31.0)
Requirement already satisfied: tqdm in /usr/local/lib/python3.10/dist-packages (from language-tool-python) (4.66.1)
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.10/dist-packages (from requests->language-tool-python
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-packages (from requests->language-tool-python) (3.4)
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.10/dist-packages (from requests->language-tool-python) (2.0
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.10/dist-packages (from requests->language-tool-python) (202
```

## Section 1: Library and Data Imports (Q1, 5 points)

- Import your libraries and join the data from both `summaries_train.csv` and `prompts_train.csv` into a single dataframe with the same structure as `use_cols`. Print the head of the dataframe. **Do not modify `use_cols`.**

```
### TODO: Load required packages
### Student's code here
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from profanity import profanity
import nltk
import re
from nltk.tokenize import sent_tokenize
import math
import nltk
import statistics
from scipy.stats import norm
nltk.download('stopwords')
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from nltk.stem import WordNetLemmatizer
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity
nltk.download('wordnet')
from collections import Counter
from sklearn.linear_model import LinearRegression
import scipy.stats as stats
from sklearn.ensemble._forest import RandomForestRegressor, GradientBoostingRegressor
from sklearn.feature_selection import SelectFromModel
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.metrics import mean_squared_error
from sklearn.metrics import r2_score
nltk.download('punkt')
import language_tool_python
import os       #importing os to set environment variable
def install_java():
  !apt-get install -y openjdk-8-jdk-headless -qq > /dev/null      #install openjdk
  os.environ["JAVA_HOME"] = "/usr/lib/jvm/java-8-openjdk-amd64"     #set environment variable
  !java -version       #check java version
install_java()
###
```

```
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
[nltk_data] Downloading package wordnet to /root/nltk_data...
[nltk_data]   Package wordnet is already up-to-date!
---------------------------------------------------------------------------
ImportError                               Traceback (most recent call last)
<ipython-input-69-b435d4910b5f> in <cell line: 25>()
     23 from sklearn.linear_model import LinearRegression
     24 import scipy.stats as stats
---> 25 from sklearn.ensemble._forest import RandomForestRegressor, GradientBoostingRegressor
     26 from sklearn.feature_selection import SelectFromModel
     27 from sklearn.preprocessing import StandardScaler

ImportError: cannot import name 'GradientBoostingRegressor' from 'sklearn.ensemble._forest' (/usr/local/lib/python3.10/dist-packages/sklearn/ensemble/_forest.py)

---------------------------------------------------------------------------
NOTE: If your import is failing due to a missing package, you can
manually install dependencies using either !pip or !apt.

To view examples of installing some common dependencies, click the
"Open Examples" button below.
---------------------------------------------------------------------------
```

OPEN EXAMPLES SEARCH STACK OVERFLOW

```
use_cols = ["student_id",
            "prompt_id",
            "text",
            "content",
            "wording",
            "prompt_question",
            "prompt_title",
            "prompt_text"
           ]
```

```
dtypes = {
        'student_id':                                   'string',
        'prompt_id':                                    'string',
        'text':                                         'string',
        'content':                                      'Float64',
        'wording':                                      'Float64',
        'prompt_question':                              'string',
        'prompt_title':                                 'string',
        'prompt_text':                                  'string',
        }
summaries=pd.read_csv("/content/drive/MyDrive/CSE519/summaries_train.csv",dtype=dtypes)
prompts=pd.read_csv("/content/drive/MyDrive/CSE519/prompts_train.csv",dtype=dtypes)
# print(summaries.shape)

idom=pd.merge(summaries,prompts,on="prompt_id")
print(idom.shape)
print(idom.columns)
# print(idom.head)
```

```
(7165, 8)
Index(['student_id', 'prompt_id', 'text', 'content', 'wording',
       'prompt_question', 'prompt_title', 'prompt_text'],
      dtype='object')
```

```
idom.head(50)
prompts.head()
prompts.prompt_title
```

```
0                   On Tragedy
1    Egyptian Social Structure
2               The Third Wave
3      Excerpt from The Jungle
Name: prompt_title, dtype: string
```

## Section 2: Features (Q2 and Q3, 25 points total)

# Prerequisite

Cleaning data -

However in our case data is already cleaned at preliminary level (has no null values).

```
## Question 2
## Calculating basic features
def count_words_simple(text):
    return len(text.split())

def word_to_set(text):
    arr=text.split()
    unique_words=set()
    for word in arr:
        unique_words.add(word)
    return unique_words

def count_words_distinct(text):#
    unique_words_set=word_to_set(text)
    return len(unique_words_set)



def count_words_advanced(text):
    cnt=len(get_valid_tokens(text))
    return cnt



#Q2 part 1
dl1=idom[["text","prompt_text"]].applymap(count_words_simple)
dl1=dl1.rename(columns={'text':'text_wc','prompt_text':'prompt_text_wc'})

#Q2 part 2
dl1[['text_uwc','prompt_text_uwc']]=idom[["text","prompt_text"]].applymap(count_words_distinct)
print(dl1)
```

```
#Q2 part 3,4,5
def common_words(row,idx=0):
    text=row.text
    if idx==1:
        prompt=row.prompt_text
    elif idx==2:
#         print("------------------>",idx)
        prompt=row.prompt_question
    elif idx==3:
        prompt=row.prompt_title

    if idx!=0:
        uw_set_text=word_to_set(text)
        uw_set_ptext=word_to_set(prompt)
        common=uw_set_text.intersection(uw_set_ptext)
        new_in_text=uw_set_text.difference(uw_set_ptext)

    return len(common)

# dl[['cmn']]=idom.apply(lambda x:common_words(x.text,x.prompt_text),axis=1)
dl2=idom.apply(lambda x:common_words(x,idx=1),axis=1)
dl1['common_uq_wc_txt_ptext']=dl2


dl3=idom.apply(lambda x:common_words(x,idx=2),axis=1)
dl1['common_uq_wc_ques']=dl3


dl4=idom.apply(lambda x:common_words(x,idx=3),axis=1)
dl1['common_uq_wc_title']=dl4
print(dl1)

## {Merging other columns with features }
dl2=idom[['content','wording','prompt_id']]
dl1=pd.merge(dl2,dl1, left_index=True, right_index=True)
print(dl1.columns)
```

```
      text_wc  prompt_text_wc  text_uwc  prompt_text_uwc
0          61             596        51              300
1         203             596       138              300
2          60             596        50              300
3          76             596        59              300
4          27             596        25              300
...       ...             ...       ...              ...
7160       33             604        30              303
7161       30             604        27              303
7162       29             604        22              303
7163       49             604        35              303
7164       59             604        42              303

[7165 rows x 4 columns]
      text_wc  prompt_text_wc  text_uwc  prompt_text_uwc  \
0          61             596        51              300
1         203             596       138              300
2          60             596        50              300
3          76             596        59              300
4          27             596        25              300
...       ...             ...       ...              ...
7160       33             604        30              303
7161       30             604        27              303
7162       29             604        22              303
7163       49             604        35              303
7164       59             604        42              303

      common_uq_wc_txt_ptext  common_uq_wc_ques  common_uq_wc_title
0                         21                  5                   1
1                         46                  9                   3
2                         29                  5                   1
3                         36                  7                   1
4                         15                  5                   1
...                      ...                ...                 ...
7160                      20                  0                   0
7161                      18                  3                   0
7162                      17                  0                   0
7163                      21                  2                   0
7164                      20                  2                   0

[7165 rows x 7 columns]
Index(['content', 'wording', 'prompt_id', 'text_wc', 'prompt_text_wc',
       'text_uwc', 'prompt_text_uwc', 'common_uq_wc_txt_ptext',
```

```
      'common_uq_wc_ques', 'common_uq_wc_title'],
     dtype='object')
```

IN this section I calculate 5 additional features apart from the ones mentioned above. They are

1. [text_ari_score] - Automated_readability_index : The automated readability index (ARI) is a readability test for English texts, designed to gauge the understandability of a text. Like the Flesch–Kincaid grade level, Gunning fog index, SMOG index, Fry readability formula, and Coleman–Liau index, it produces an approximate representation of the US grade level needed to comprehend the text. Why this? Because its simple and doesn't requires syllable i.e phonetic interpretation.
2. [summary_len_percent] Summary Length percentage: This is a simple percentage of length of summary by length of actual prompt_text. Ideally it should be neither too long or nor too short. So between 5-15 %
3. [text_similarity_score] - Text Similarity Score: This is the norm of cosine similarity betweeen 2 text .While this is a fairly good estimate of text similarity it has some limitations It converts text to tokens and then into further vectors removing the grammatical construct that language imposes, which is necessary for a good essay.

To counter this, I chose another parameter albeit much more time-consuming to incorporate becasue of environmental issues. Both these libraries take more time as they rely on remote databases and have to contact it via remote API calls for a given text.

4. [grammar_correction_cnt] It uses language_tool_python, a python library that provides grammatical (spelling + grammar) error counts in a text.
5. [has_bad_words] - Type Boolean , later converted to One hot encoded columsn to take care of th ecategorical nature This uses profanity library to identify whether the given text has any sort of profane words. I used a very simple implementationof Yes or no to cater the same.

```
## !! Expected Runtime between 8 min and 12 min !!

## Calculating additional features -------------------

# Feature 1 -Automated_readability_index
def calculate_ari_score(text):
    space_cnt=text.count(" ")
    character_cnt=len(re.findall('[0-9A-z]',text))
    sentences_arr = sent_tokenize(text)
    sentence_cnt=len(sentences_arr)
    ari=(4.71*(character_cnt/space_cnt))+(0.5*(space_cnt/sentence_cnt))-21.43
    return math.ceil(ari)

dl5=idom[["text"]].applymap(calculate_ari_score)
dl5.rename(columns={'text':'text_ari'})
dl1['text_ari_score']=(dl5)
# print(dl1)

# Feature 2 -Summary Length Percentage with respect to Prompt Text
# Length of summary with respect to prompt length

def calculate_summary_size(row):
    text=row.text
    summ_text=row.prompt_text
    summary_length=len(row.text)
    prompt_length=len(row.prompt_text)
    val=(summary_length/prompt_length)*100
    return summary_length,prompt_length,val

dl6=idom.apply(lambda x:calculate_summary_size(x),axis=1)
dl1[['summary_len','prompt_length','summary_len_percent']]=list(dl6)
# print(dl1)

# Feature 3 -Whether the text contains bad words according to profanity
def contains_badwords(text):
    ans=profanity.contains_profanity(text)
    return ans

df=idom["text"].apply(contains_badwords)
dl1["has_bad_words"]=df
print(dl1)

# Feature 4 - Text similarity based on norm of cosine similarity matrix
def text_similarity(row):
    text1=row.text
    text2=row.prompt_text
```

```
        # Tokenize and lemmatize the texts
        tokens1 = word_tokenize(text1)
        tokens2 = word_tokenize(text2)
        lemmatizer = WordNetLemmatizer()
        tokens1 = [lemmatizer.lemmatize(token) for token in tokens1]
        tokens2 = [lemmatizer.lemmatize(token) for token in tokens2]

        # Remove stopwords
        stop_words = stopwords.words('english')
        tokens1 = [token for token in tokens1 if token not in stop_words]
        tokens2 = [token for token in tokens2 if token not in stop_words]

        # Create the TF-IDF vectors
        vectorizer = TfidfVectorizer()
        vector1 = vectorizer.fit_transform(tokens1)
        vector2 = vectorizer.transform(tokens2)

        # Calculate the cosine similarity
        similarity = cosine_similarity(vector1, vector2)

        return similarity

    df=idom.apply(lambda x:text_similarity(x),axis=1)
    dl1["text_similarity"]=df

    def perc_sim_from_cosine(x):
          val=np.linalg.norm(x.text_similarity)
          return val

    df=dl1.apply(lambda x: perc_sim_from_cosine(x),axis=1)
    dl1["text_similarity_score"]=df
    # print(dl1)

    # Feature 5 - Number of grammatical &/or spelling errros identified by language_tool_python
    # this step requires java setup hence the dependency and does take roughly 5+-2 min since every text has to verified via internal api call

    cnt=0
    tool = language_tool_python.LanguageTool('en-US')
    def grammar_correction_cnt(row):
    #     text = u'A sentence with a error in the Hitchhiker’s Guide tot he Galaxy'
        matches = tool.check(row.text)
        global cnt
        cnt+=1
        if cnt%1000==0:
          print(cnt)
        return (len(matches))

    df=idom.apply(lambda x: grammar_correction_cnt(x),axis=1)
    dl1["grammar_correction_cnt"]=df
    print(dl1)
```

```
           content   wording prompt_id  text_wc  prompt_text_wc  text_uwc  \
    0     0.205683  0.380538    814d6b       61             596        51
    1     3.272894  3.219757    814d6b      203             596       138
    2     0.205683  0.380538    814d6b       60             596        50
    3     0.567975  0.969062    814d6b       76             596        59
    4    -0.910596 -0.081769    814d6b       27             596        25
    ...        ...       ...       ...      ...             ...       ...
    7160 -0.981265   -1.5489    39c16e       33             604        30
    7161 -0.511077 -1.589115    39c16e       30             604        27
    7162 -0.834946 -0.593749    39c16e       29             604        22
    7163  -0.15746 -0.165811    39c16e       49             604        35
    7164  -0.39331  0.627128    39c16e       59             604        42

          prompt_text_uwc  common_uq_wc_txt_ptext  common_uq_wc_ques  \
    0                 300                      21                  5
    1                 300                      46                  9
    2                 300                      29                  5
    3                 300                      36                  7
    4                 300                      15                  5
    ...               ...                     ...                ...
    7160              303                      20                  0
    7161              303                      18                  3
```

```
7162            303                     17                0
7163            303                     21                2
7164            303                     20                2

      common_uq_wc_title  text_ari_score  summary_len  prompt_length  \
0                      1               9        346.0         3566.0
1                      3               9       1225.0         3566.0
2                      1              11        345.0         3566.0
3                      1              15        451.0         3566.0
4                      1               7        145.0         3566.0
...                  ...             ...          ...            ...
7160                   0              17        180.0         3364.0
7161                   0              15        163.0         3364.0
7162                   0              13        150.0         3364.0
7163                   0              15        297.0         3364.0
7164                   0               8        294.0         3364.0

      summary_len_percent  has_bad_words
0                9.702748          False
1               34.352215          False
2                9.674706          False
3               12.647224          False
4                4.066181          False
...                   ...            ...
7160             5.350773          False
7161             4.845422          False
7162             4.458977          False
7163             8.828775          False
7164             8.739596          False

[7165 rows x 15 columns]
```

```
# # Feature 5 - Number of grammatical &/or spelling errros identified by language_tool_python
# ## this step requires java setup hence the dependency and does take roughly 5+-2 min since every text has to verified via internal api call

# cnt=0
# tool = language_tool_python.LanguageTool('en-US')
# def grammar_correction_cnt(row):
# #     text = u'A sentence with a error in the Hitchhiker’s Guide tot he Galaxy'
#     matches = tool.check(row.text)
#     global cnt
#     cnt+=1
#     if cnt%1000==0:
#       print(cnt)
#     return (len(matches))

# df=idom.apply(lambda x: grammar_correction_cnt(x),axis=1)
# dl1["grammar_correction_cnt"]=df
# print(dl1)
```

```
7161            303                     18                3
7162            303                     17                0
7163            303                     21                2
7164            303                     20                2
```

```
0     [[0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,...
1     [[0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,...
2     [[0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,...
3     [[0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,...
4     [[0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,...
...                                                 ...
7160  [[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,...
7161  [[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,...
7162  [[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,...
7163  [[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,...
7164  [[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,...

      text_similarity_score  grammar_correction_cnt
0                  7.681146                       3
1                 16.822604                      15
2                 10.770330                       3
3                 10.816654                       4
4                  6.324555                       3
...                     ...                     ...
7160               6.000000                       0
7161               4.358899                       3
7162               6.204837                       2
7163               6.855655                       1
7164               6.782330                       1

[7165 rows x 18 columns]
```

In this section we try to factor in (factor out )various filler texts, by getting only valid tokens out of the text in all texts including ["text","prompt_text","prompt_question","prompt_title"] This is because we don't want special characters like ["",",``,',] etc. affect our data points. We do this by

1. Filtering out special characters
2. Tokenising words
3. Lemmatsing words like [counting play ,played,plays as 1 etc.]
4. Removing stopwords like [a ,an, of ]
5. I removed some additional words which were not included in stopwords dictionary
6. I also removed single character words[ because they rarely communnicate any significant meaning and also ] because the library was skipping few unncessary characters.
7. I didn't convert the text into lowercase as it might hide some significant spelling issues or language errors, which i feel will affect score for wording.Also I intend to identify such issues using grammar_correction_cnt parameter.

We apply this logic to all text data columns so it should have a uniform impact on all columns, (Thought for improvement: except in essays where they are siginificantly high , where it has affected grading).

```
## !! Running time  4 mins !! ---> reduced to <1 min
### advanced filtering  For first 5 basic parameters

# Preprocessing data --cleaning data
# Util function to clean text
def get_valid_tokens(text1,threshold=2):
    # Strip special characters
    bad_chars=[';', ':', '!', "*", "'",'"',"`",'.',",",]
    text1=''.join(letter for letter in text1 if not  letter in bad_chars)
    # Tokenize and lemmatize the texts
    tokens1 = word_tokenize(text1)

    lemmatizer = WordNetLemmatizer()
    tokens1 = [lemmatizer.lemmatize(token) for token in tokens1]
    # Remove stopwords
    stop_words = stopwords.words('english')
    stop_words2=["This"]
    tokens1 = [token for token in tokens1 if token not in stop_words]
    tokens1 = [token for token in tokens1 if token not in stop_words2]
    #remove_single_char_func
    final=[word for word in tokens1 if len(word) > threshold]
    return final

def clean_data(text):
    tokens=get_valid_tokens(text)
    cnt=len(tokens)
    text=" ".join(tokens)
    return text

trimmed_idom=idom[["text","prompt_text","prompt_question","prompt_title"]].applymap(clean_data)
```

```
#Q2 part 1
adv_dl1=trimmed_idom[["text","prompt_text"]].applymap(count_words_simple)
adv_dl1=adv_dl1.rename(columns={'text':'text_wc','prompt_text':'prompt_text_wc'})

#Q2 part 2
adv_dl1[['text_uwc','prompt_text_uwc']]=idom[["text","prompt_text"]].applymap(count_words_distinct)
# print(adv_dl1)

#Q2 part 3,4,5
# dl[['cmn']]=idom.apply(lambda x:common_words(x.text,x.prompt_text),axis=1)
dl2=idom.apply(lambda x:common_words(x,idx=1),axis=1)
adv_dl1['common_uq_wc_txt_ptext']=dl2


dl3=idom.apply(lambda x:common_words(x,idx=2),axis=1)
adv_dl1['common_uq_wc_ques']=dl3


dl4=idom.apply(lambda x:common_words(x,idx=3),axis=1)
adv_dl1['common_uq_wc_title']=dl4
print(adv_dl1)

## {Merging other columns with features }
dl2=idom[['content','wording','prompt_id']]
adv_dl1=pd.merge(dl2,adv_dl1, left_index=True, right_index=True)
print(adv_dl1.columns)
```

```
      text_wc  prompt_text_wc  text_uwc  prompt_text_uwc  \
0          39             315        51              300
1         119             315       138              300
2          33             315        50              300
3          40             315        59              300
4          16             315        25              300
...       ...             ...       ...              ...
7160       17             284        30              303
7161       15             284        27              303
7162       13             284        22              303
7163       27             284        35              303
7164       32             284        42              303

      common_uq_wc_txt_ptext  common_uq_wc_ques  common_uq_wc_title
0                         21                  5                   1
1                         46                  9                   3
2                         29                  5                   1
3                         36                  7                   1
4                         15                  5                   1
...                      ...                ...                 ...
7160                      20                  0                   0
7161                      18                  3                   0
7162                      17                  0                   0
7163                      21                  2                   0
7164                      20                  2                   0

[7165 rows x 7 columns]
Index(['content', 'wording', 'prompt_id', 'text_wc', 'prompt_text_wc',
       'text_uwc', 'prompt_text_uwc', 'common_uq_wc_txt_ptext',
       'common_uq_wc_ques', 'common_uq_wc_title'],
      dtype='object')
```

## Section 3: Content and Wording (Q4, 10 points)

As expected,the correlation between Content and wording score is positive and high.It can be inferred that a student who writes good content will generally have worked harder and thus use better wording. It can also be assumed that generally grades do correlate as in people who work hard in 1 class tend to work harder in other classes as well. Based on it we can say that this correlation is justified.

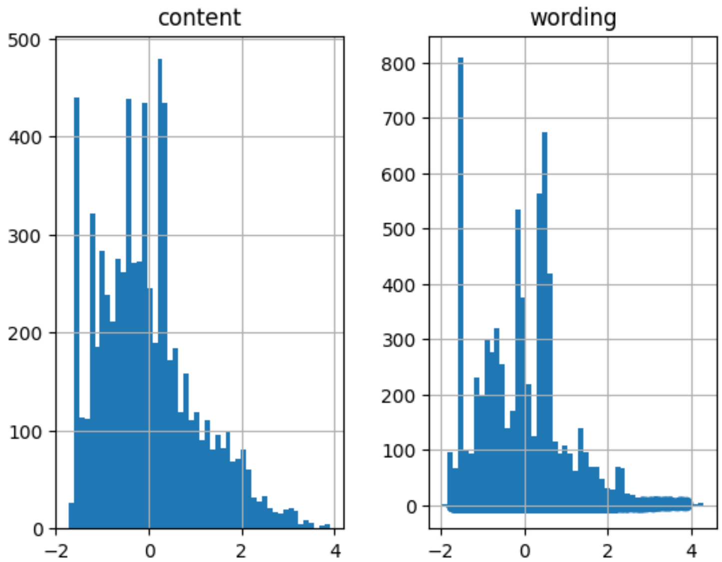
Note: I noticed that "content" and "wording" are normalized. as seen from their distribution curves. It is essential to get decent results to normalise the scores.

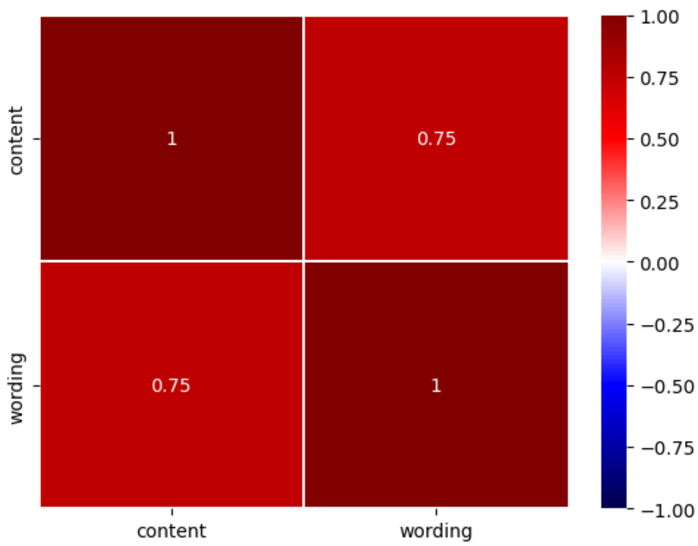
I have calculated common words for eac of the prompt separately.It is because I feel that each prompt_text has its own unique nature /style. FOr eg. the treatment required by a summary 'On Tragedy' is different than a more analytical 'Egyptian Social Structure' .

```
score=idom[['content', 'wording']]
hists=score.hist(bins=50)
```

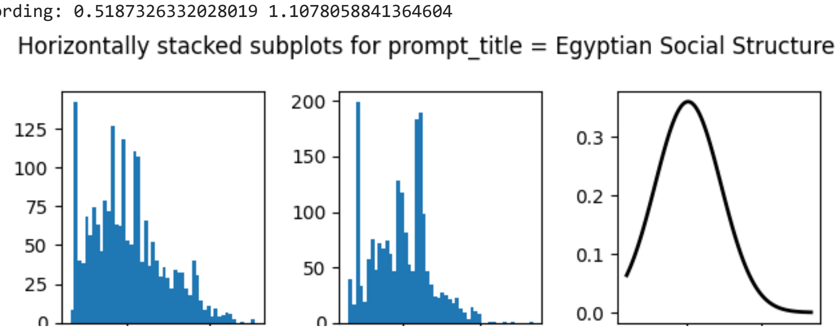
```
score.corr()
print(score)
score.describe()
plt.scatter(score['content'],score['wording'])
plt.show()

sns.heatmap(score.corr(), vmin=-1, vmax=1, cmap='seismic', linewidths=0.2, annot=True)
plt.show()



def plots(data,prompt_title):
    mu1,sd1= statistics.mean(data.content),statistics.stdev(data.content)
    mu2,sd2 = statistics.mean(data.wording),statistics.stdev(data.wording)
    print("content:",mu1,sd1)
    print("wording:",mu2,sd2)
    fig, ((ax1, ax2,ax5), (ax3, ax4,ax6))  = plt.subplots(2,3)
    content,wording='content','wording'
    fig.suptitle('Horizontally stacked subplots for prompt_title = '+prompt_title)
    fig.tight_layout(pad=1.0)
    ax1.hist(data[content],bins=50)
    xmin, xmax = ax1.get_xlim()
    x = np.linspace(xmin, xmax, 100)
    p = norm.pdf(x, mu1, sd1)
    ax5.plot(x,p,'k',linewidth=2)
    ax5.set_xlabel(content)
    ax1.set_xlabel(content)
    ax2.hist(data[wording],bins=50)
    ax2.set_xlabel(wording)
    ax3.scatter(data[content],data[wording])
    ax3.set_xlabel(content)
    ax3.set_ylabel(wording)
    ax4.scatter(data[wording],data[content])
    ax4.set_xlabel(wording)
    ax4.set_ylabel(content)
    xmin, xmax = ax3.get_xlim()
    x = np.linspace(xmin, xmax, 100)
    p = norm.pdf(x, mu2, sd2)
    ax6.plot(x,p,'k',linewidth=2)
    ax6.set_xlabel(wording)

for row in set(idom.prompt_id):
    temp=idom[idom.prompt_id==row]
    pt=prompts[prompts.prompt_id==row]
    pt_val=pt.iloc[0]['prompt_title']
    plots(temp,pt_val)

# plt.subplot_tool()
plt.show()
```

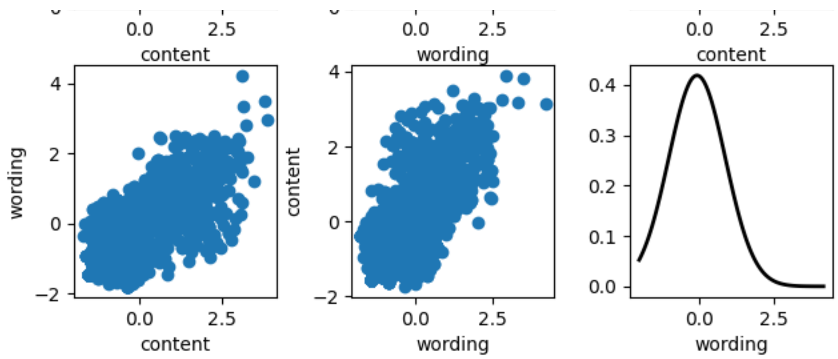
```
       content   wording
0     0.205683  0.380538
1     3.272894  3.219757
2     0.205683  0.380538
3     0.567975  0.969062
4    -0.910596 -0.081769
...        ...       ...
7160 -0.981265   -1.5489
7161 -0.511077 -1.589115
7162 -0.834946 -0.593749
7163  -0.15746 -0.165811
7164  -0.39331  0.627128

[7165 rows x 2 columns]
```

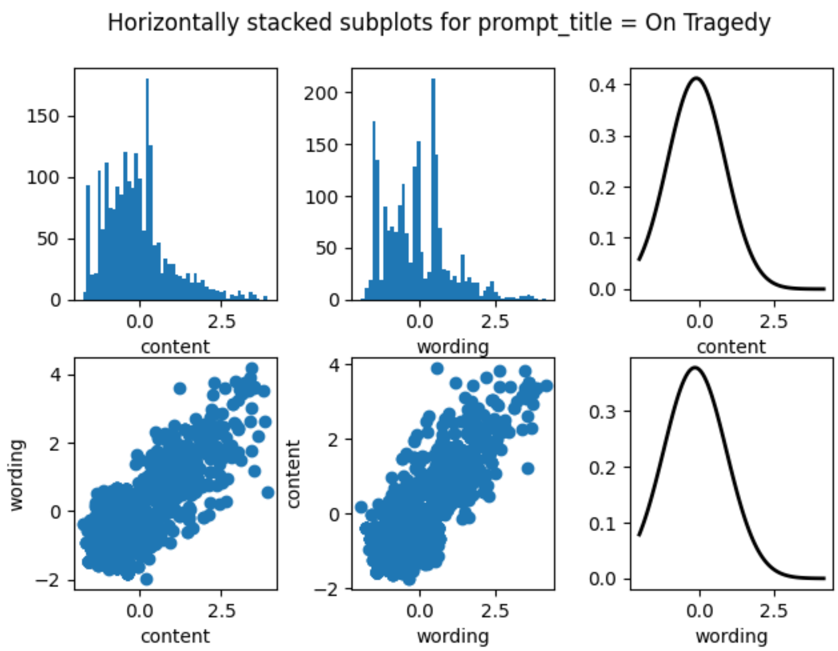


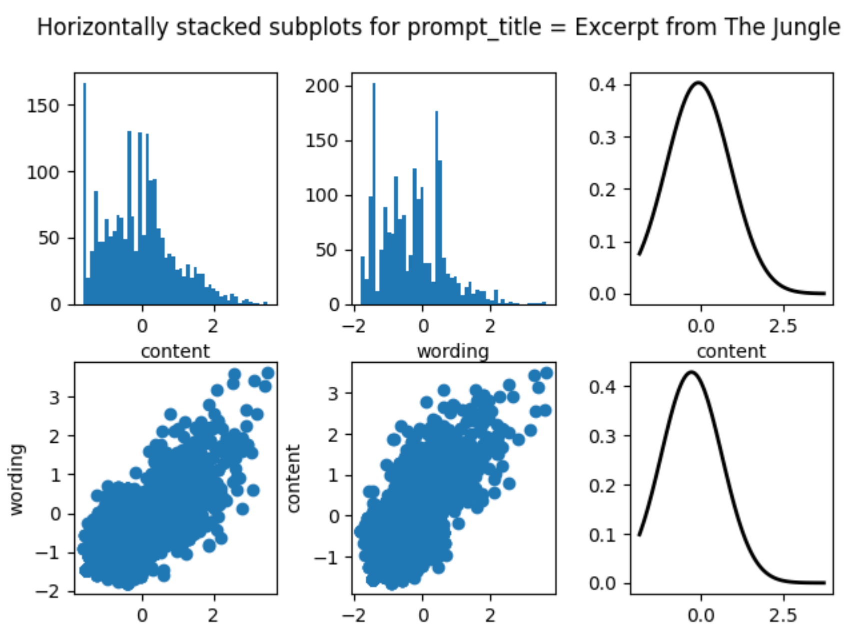




```
content: 0.04957931058413491 1.106128635106322
wording: -0.06854168562860835 0.9527077504423439
content: -0.09545655978220603 0.9697733354136258
wording: -0.1407492420899874 1.0556950481512974
content: -0.0879058421378715 0.9902711894077203
wording: -0.2990231732033524 0.9302701095279209
content: 0.15030633033483762 1.1241576869269378
wording: 0.5187326332028019 1.1078058841364604
```





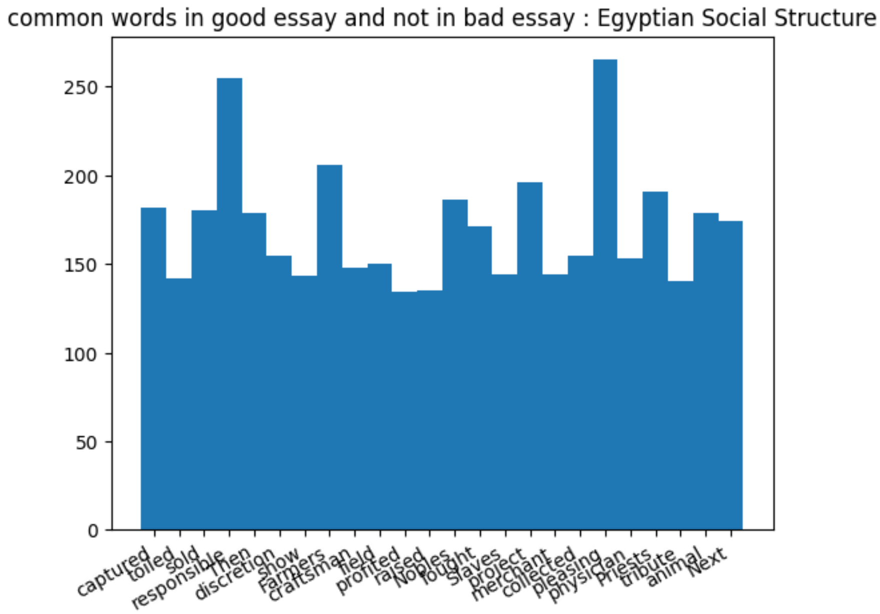




## Section 4: Words in Good and Bad Essays (Q5, 10 points)

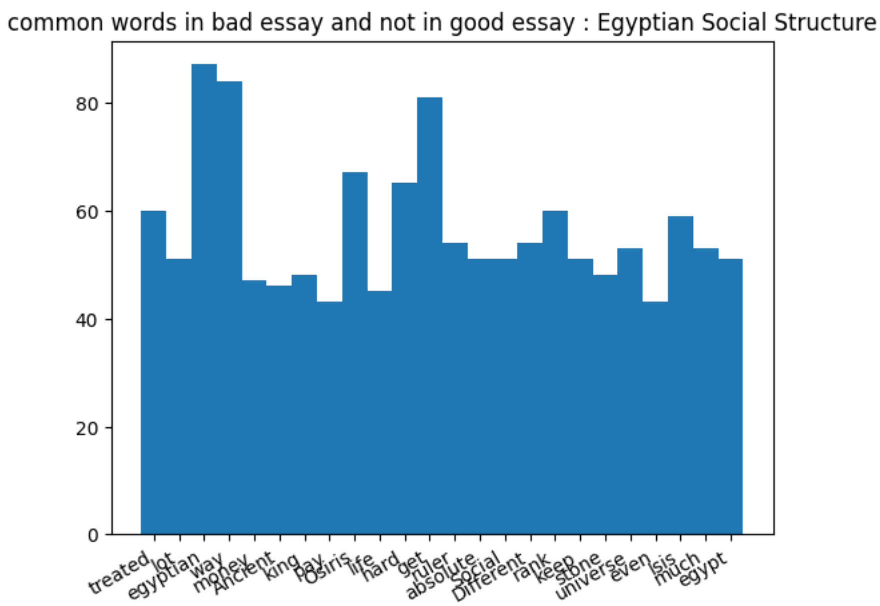
```
## Overrrepeseanted words
# q5=idom.sort_values(by=['content','wording'], ascending=[False,False])
def get_valid_tokens(text1,threshold=2):
    # Strip special characters
    bad_chars=[';', ':', '!', "*", "'",'"',"`",'.',",",]
    text1=''.join(letter for letter in text1 if not  letter in bad_chars)
    # Tokenize and lemmatize the texts
```

```
    tokens1 = word_tokenize(text1)

    lemmatizer = WordNetLemmatizer()
    tokens1 = [lemmatizer.lemmatize(token) for token in tokens1]
    # Remove stopwords
    stop_words = stopwords.words('english')
    stop_words2=["This"]
    tokens1 = [token for token in tokens1 if token not in stop_words]
    tokens1 = [token for token in tokens1 if token not in stop_words2]
    #remove_single_char_func
    final=[word for word in tokens1 if len(word) > threshold]
    return final


counter=Counter()
def common_words(row):
    text=row.text
    tokens=get_valid_tokens(text)
    local=Counter(get_valid_tokens(text))
    global counter
    counter=counter+local
    return local

grps=set(idom.prompt_id)
q51=idom.groupby("prompt_id")
for grp in grps:
    pt=prompts[prompts.prompt_id==grp]
    pt_val=pt.iloc[0]['prompt_title']
    print( " For grp -",grp, pt_val)
    tempq5=q51.get_group(grp)
    q52=tempq5.sort_values(by=['content','wording'], ascending=[False,False])
    sample_size=math.ceil(len(q52)/2)
    best=q52.head(sample_size)
    worst=q52.tail(sample_size)
    global counter
    counter=Counter()
    best.apply(lambda x:common_words(x),axis=1)
    w1=counter.most_common(100)
    top_good_dict=dict(w1)
    top_good=set(top_good_dict.keys())
    counter=Counter()
    worst.apply(lambda x:common_words(x),axis=1)
    w2=counter.most_common(100)
    top_bad_dict=dict(w2)
    top_bad=set(top_bad_dict.keys())

    words=top_good-top_bad
    final_dict={}
    for word in words:
        final_dict[word]=top_good_dict[word]
    title="common words in good essay and not in bad essay : "+pt_val
    print(title+":{0}".format(final_dict))
    plt.bar(final_dict.keys(), final_dict.values(),width=1)
    plt.xticks(rotation=30, ha='right')
    plt.title(title)
    plt.show()
    bad_words=top_bad-top_good
    bad_final_dict={}
    for word in bad_words:
        bad_final_dict[word]=top_bad_dict[word]
    title="common words in bad essay and not in good essay : "+pt_val
    print(title+":{0}".format(bad_final_dict))
    plt.bar(bad_final_dict.keys(), bad_final_dict.values(),width=1)
    plt.xticks(rotation=30, ha='right')
    plt.title(title)
    plt.show()
```

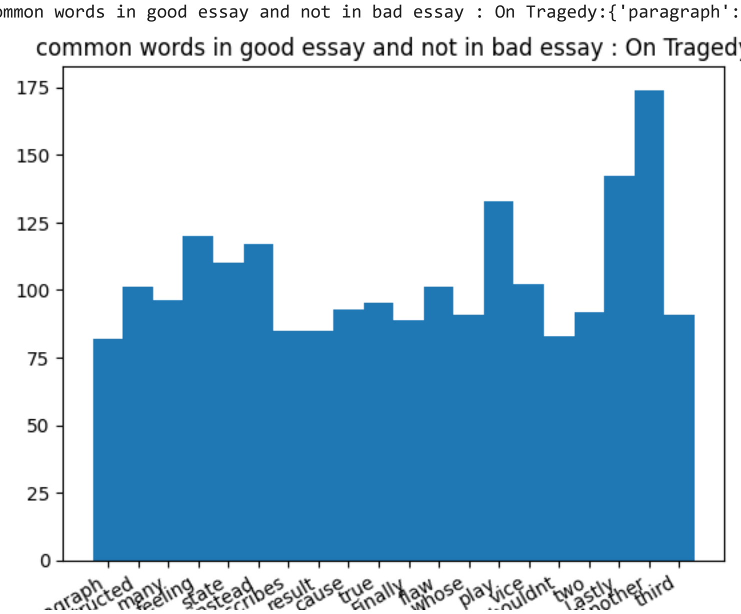
```
For grp - 3b9047 Egyptian Social Structure
common words in good essay and not in bad essay : Egyptian Social Structure:{'captured': 182, 'toiled': 142, 'sold': 180, 'responsib
```



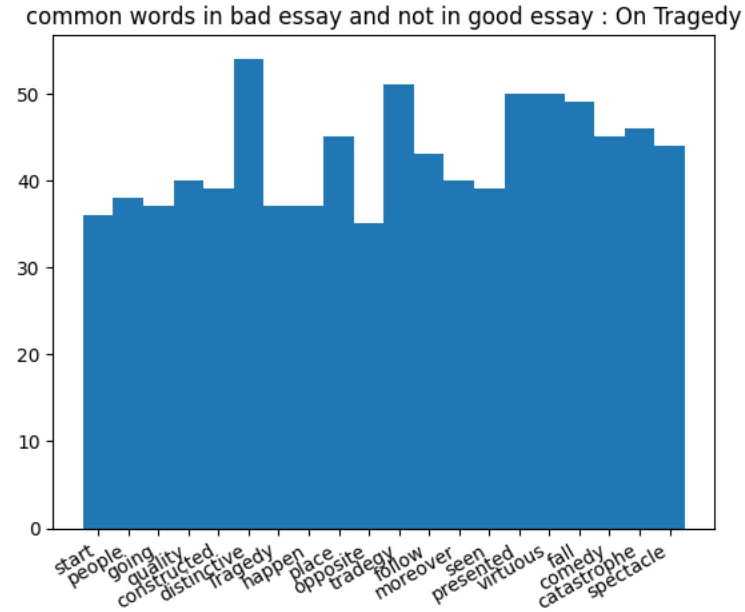

```
common words in bad essay and not in good essay : Egyptian Social Structure:{'treated': 60, 'lot': 51, 'egyptian': 87, 'way': 84, 'm
```



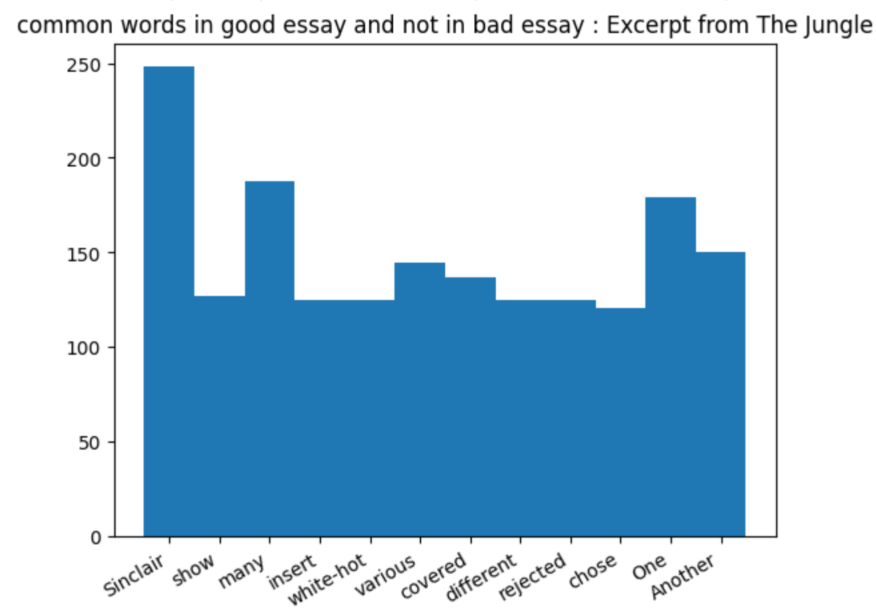

```
For grp - 39c16e On Tragedy
common words in good essay and not in bad essay : On Tragedy:{'paragraph': 82, 'well-constructed': 101, 'many': 96, 'feeling': 120,
```



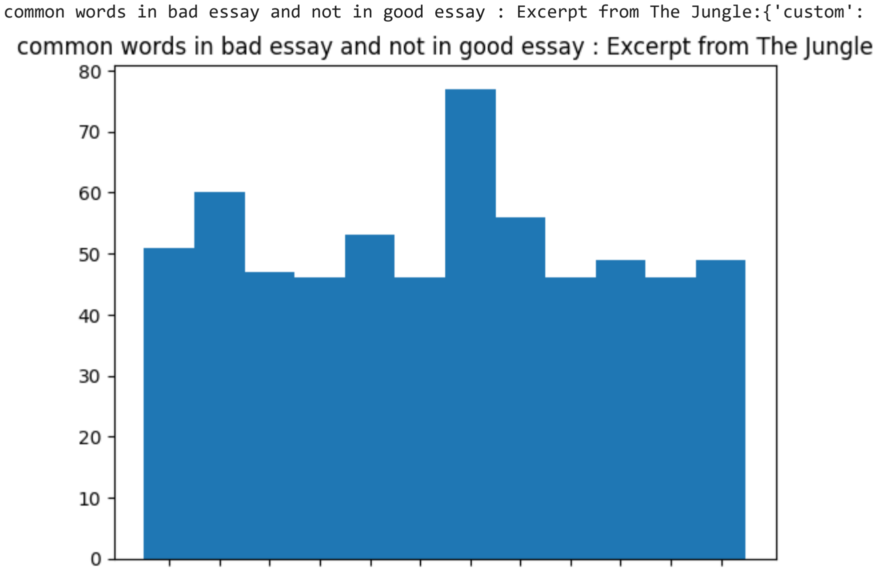
common words in bad essay and not in good essay : On Tragedy:{'start': 36, 'people': 38, 'going': 37, 'quality': 40, 'constructed':

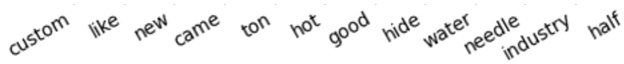



For grp - ebad26 Excerpt from The Jungle
common words in good essay and not in bad essay : Excerpt from The Jungle:{'Sinclair': 248, 'show': 127, 'many': 188, 'insert': 125,

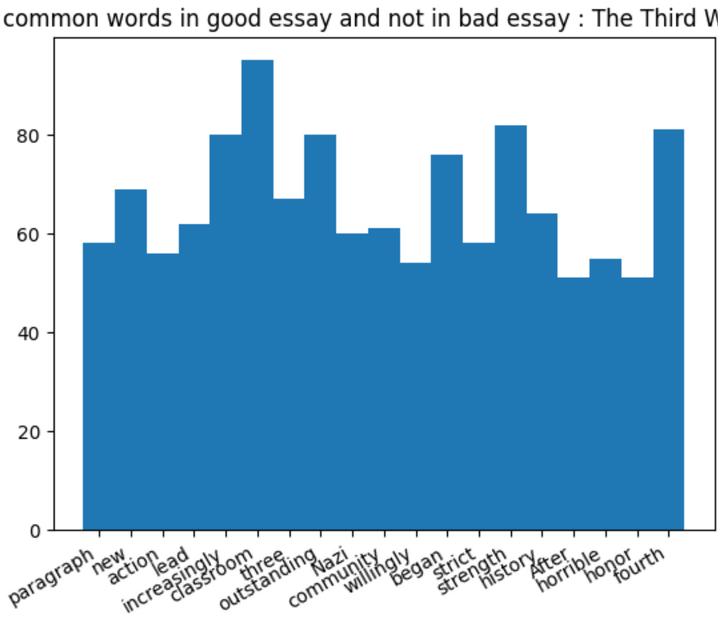



common words in bad essay and not in good essay : Excerpt from The Jungle:{'custom': 51, 'like': 60, 'new': 47, 'came': 46, 'ton': 5

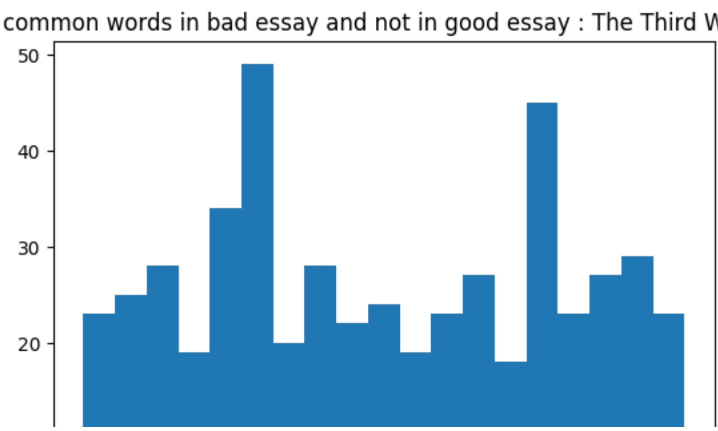





```
For grp - 814d6b The Third Wave
common words in good essay and not in bad essay : The Third Wave:{'paragraph': 58, 'new': 69, 'action': 56, 'lead': 62, 'increasingly
```




```
common words in bad essay and not in good essay : The Third Wave:{'going': 23, 'said': 25, 'see': 28, 'lot': 19, 'didnt': 34, 'fast'
```




## Section 5: Three Interesting Plots (Q6, 15 points)

### Preliminary Inferences

Here are some inferences we can make:

1. The content and wording scores could be indicative of the quality of the summaries. Higher scores in these columns might suggest that the summary is well-written and accurately captures the main points of the prompt.
2. The range of scores for both content and wording suggests that there is a variety in the quality of summaries. This could be due to differences in understanding of the prompt, writing skills, or even effort put into the task by different students.
3. The same prompt_id can have varying content and wording scores. This indicates that students may interpret and summarize the same prompt in different ways, leading to varying levels of quality in their summaries.
4. Some prompts might be more challenging to summarize than others, as indicated by lower average scores. For example, prompt ebad26 has a negative content score, suggesting that it might be difficult for students to summarize effectively.
5. The data could be used to identify areas where students may need additional support or instruction, such as summarizing information or improving their writing skills.

### 1) Interesting Correlation or absence of correlation

a) Absence of any significant correlation between ARI_Score with Content and ARI_Score with Wording is surprising. [As highlighted in Correlation matrix ] b)Usually one would expect a positive correlation between ARI_Score and wording but there is a negative correlation which means the grader intend to have a general simplicity. c) Model insensitivity to feature parameters Through the process I realised even seemingly logical relations need not be data driven. Self bias can easily creep in based on feature selection.

## 2) Interesting outlier

a) A guy with highest Ari_score of 235 has a summary length percentage of 75% which means he actually wrote a lot of text ( I feel sorry for him) It may seem like he just copy pasted the entire text and merely rewrote a few lines, more like reverse engineer a summary. But the text_similarity shows that the text is significantly different His z_score for [text_similarity_score ] was 1.1 His score is average in both Content andn wording , which means maybe the teacher penalised him for writing too long.

## 3) The bar plots on content and wording tell us that generic words usually lead to lower score.

a) On Tragedy:

Priority to Qualitative aspect: This essay deals with [feeling , emotions] as reflecetd by popular words. Priorty to Cohesive sentences: sentence construction is clearly a popular trait in this essay as [ Another, Lastly ] like words are popular in well graded essays. common words in bad essay share no signiifcant insight.

b) Egyptian Social Structure

Priority to Analytical and Objective aspect: This essay deals with more objective analysis as reflecetd by popular words.['captured': 182, 'toiled': 142, 'sold': 180, 'responsible': 255, 'Then': 179, 'discretion': 155, 'show': 143, 'physician': 153, 'Priests': 191, 'tribute': 140] The bad words indicate thaat those who kept the summary more historical or mythological or Person oriented, missed the analytical judgment and thus scored less.

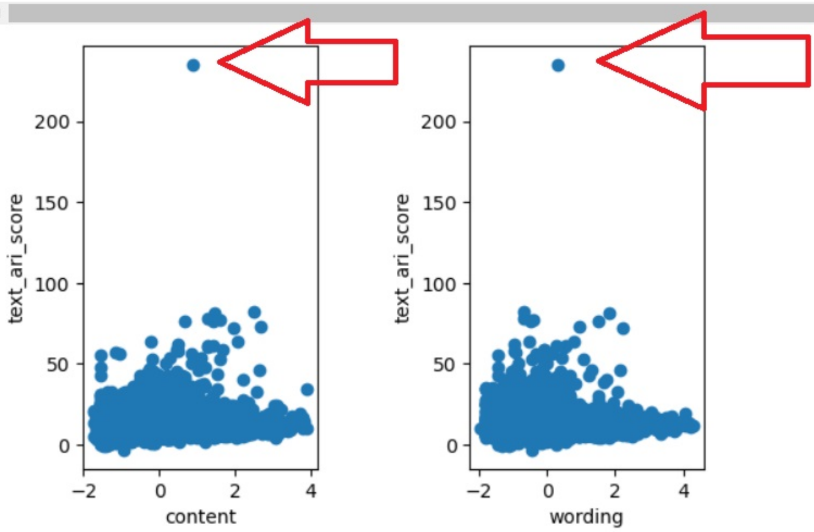
c) The Third Wave

->Easy Essay or Smarter kids or Easy Checking Whatever maybe the reason this essay students got the best scores. It seems liek a historical text summary As a result, Factual Information and specific content words seem to be more appreciated compared to generic words in poorly scored essays.

d) Excerpt from The Jungle

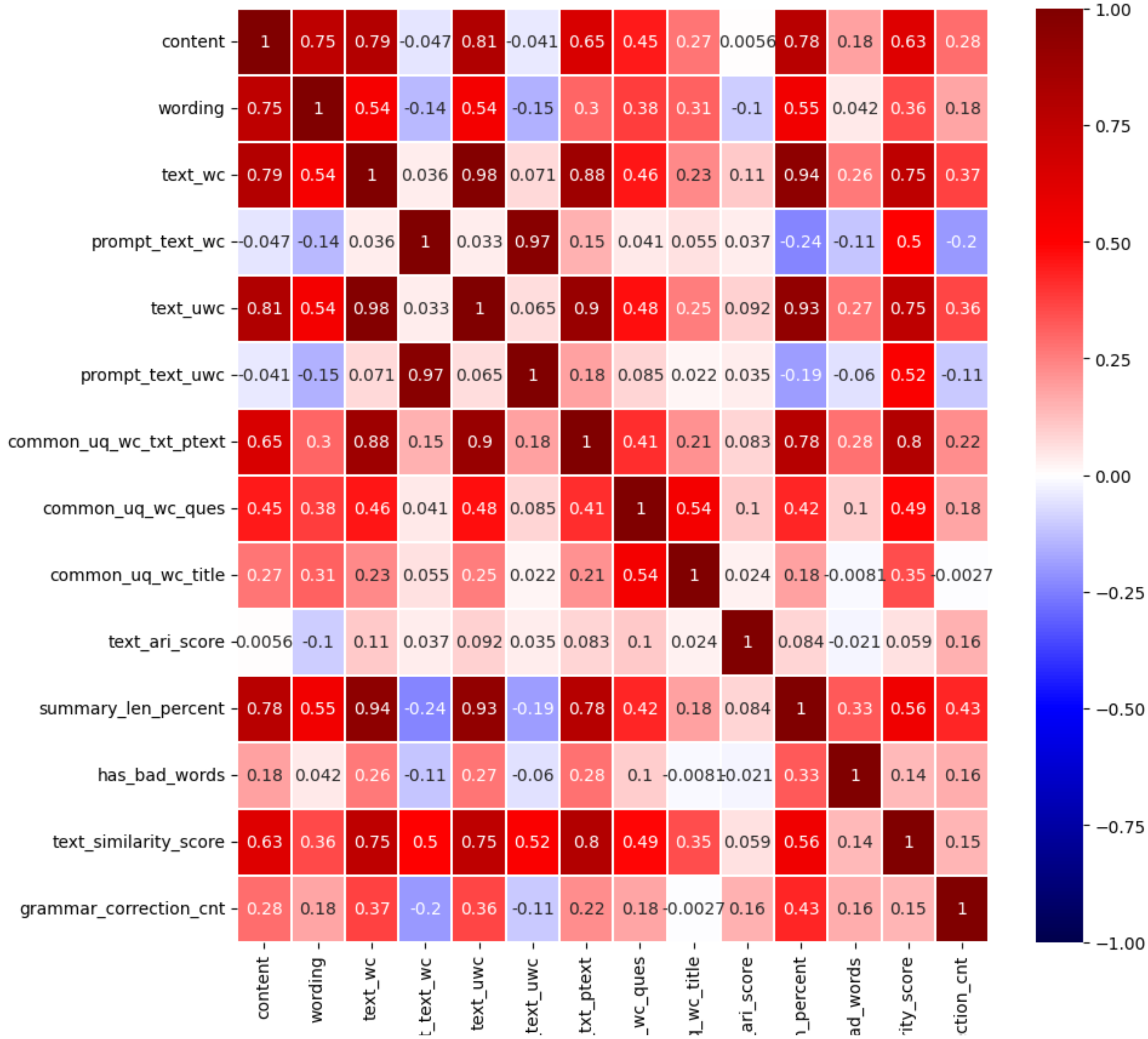
Lowest Scoring essay It seems the children couldn't get the context of the essay as many good graded essy also have a lot of filler words [highest among all prompts] . A lot of student maybe have mentioned similar words and sentences leading to a very quite similar frequency bar graph for pouplar words in good essays.

| | content | wording | prompt_id | text_wc | prompt_text_wc | text_uwc | prompt_text_uwc | text_ari_score | summary_len | prompt_length | summary_len_percent |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 3800 | 0.903185 | 0.295562 | 3b9047 | 471 | 550 | 210 | 319 | 235 | 2513.0 | 3345.0 | 75.127055 |




```
## 1a) presence and absence of corrleation
fig, ax = plt.subplots(figsize=(10,10))
dl_corr=dl1[['content', 'wording',  'text_wc', 'prompt_text_wc',
       'text_uwc', 'prompt_text_uwc', 'common_uq_wc_txt_ptext',
       'common_uq_wc_ques', 'common_uq_wc_title', 'text_ari_score',
        'summary_len_percent', 'has_bad_words','text_similarity_score', 'grammar_correction_cnt']]
sns.heatmap(dl_corr.corr(), vmin=-1, vmax=1, cmap='seismic', linewidths=0.2, annot=True,ax=ax)
```

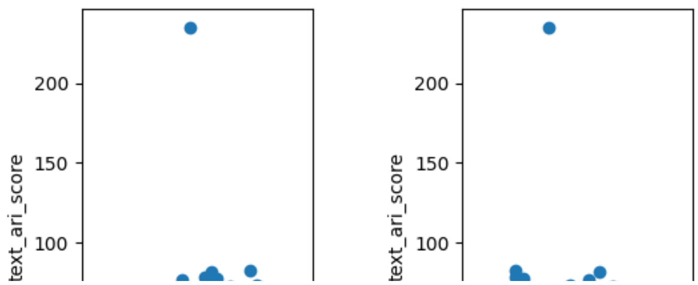
<Axes: >



```
# Observation 2a

dl3=dl1
dl3=dl3.sort_values(by=['content','wording','prompt_id'], ascending=[False,False,False])

print(len(set(dl1.text_similarity_score)))
print(dl1.text_similarity_score.max(),dl1.text_similarity_score.min())
fig, (ax,bx) = plt.subplots(1,2)
fig.tight_layout(pad=5.0)
ax.scatter(dl3.content,dl3.text_ari_score)
ax.set_xlabel('content')
ax.set_ylabel('text_ari_score')
bx.scatter(dl3.wording,dl3.text_ari_score)
bx.set_xlabel('wording')
bx.set_ylabel('text_ari_score')
z_score=stats.zscore(dl3.text_similarity_score)
print(z_score.loc[3800])
dl3[dl3.text_ari_score==235]
# dl3
```

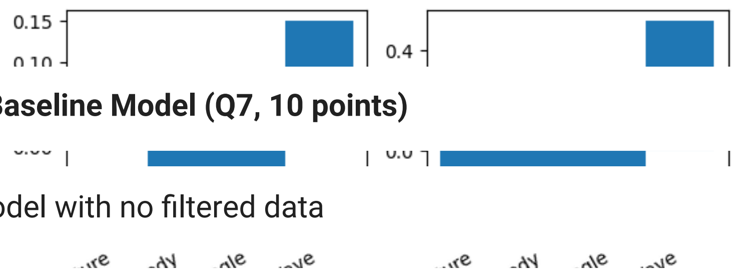
```
1783
32.97282594915417 1.0
1.1001604128197195
```

| | content | wording | prompt_id | text_wc | prompt_text_wc | text_uwc | prompt_text_uwc | common_uq_wc_txt_ptext | common_uq_wc_ques | commo |
|---|---|---|---|---|---|---|---|---|---|---|
| **3800** | 0.903185 | 0.295562 | 3b9047 | 471 | 550 | 210 | 319 | 59 | 6 | |




```
grps=set(idom.prompt_id)
q51=idom.groupby("prompt_id")
mean_arr_content,mean_arr_wording={},{}
perc75_class_content={}
perc75_class_wording={}
for grp in grps:
    pt=prompts[prompts.prompt_id==grp]
    pt_val=pt.iloc[0]['prompt_title']
    print( " For grp -",grp, pt_val)
    tempq5=q51.get_group(grp)
    q52=tempq5.sort_values(by=['content','wording'], ascending=[False,False])
    mean_arr_content[pt_val]=q52['content'].mean()
    mean_arr_wording[pt_val]=q52['wording'].mean()
    perc75_class_content[pt_val]=np.percentile(q52['content'], 75, axis =0)
    perc75_class_wording[pt_val]=np.percentile(q52['wording'], 75, axis =0)

print(mean_arr)

# plt.hist(perc75_class,bins=10)
fig, ((ax3,ax4),(ax1, ax2))  = plt.subplots(2,2)
ax3.bar(mean_arr_content.keys(), mean_arr_content.values(),width=1)
ax3.set_xticklabels(mean_arr_content.keys(), rotation=30, ha='right')
ax4.bar(mean_arr_wording.keys(), mean_arr_wording.values(),width=1)
ax4.set_xticklabels(mean_arr_wording.keys(), rotation=30, ha='right')
ax1.bar(perc75_class_content.keys(), perc75_class_content.values(),width=1)
ax1.set_xticklabels(perc75_class_content.keys(), rotation=30, ha='right')
ax2.bar(perc75_class_wording.keys(), perc75_class_wording.values(),width=1)
ax2.set_xticklabels(perc75_class_wording.keys(), rotation=30, ha='right')
# plt.title(title)
plt.show()
```

```
 For grp - 3b9047 Egyptian Social Structure
 For grp - 39c16e On Tragedy
 For grp - ebad26 Excerpt from The Jungle
 For grp - 814d6b The Third Wave
{'Egyptian Social Structure': content    0.049579
wording   -0.068542
dtype: float64, 'On Tragedy': content   -0.095457
wording   -0.140749
dtype: float64, 'Excerpt from The Jungle': content   -0.087906
wording   -0.299023
dtype: float64, 'The Third Wave': content    0.150306
wording    0.518733
dtype: float64}
<ipython-input-135-5bcc6f65d002>:22: UserWarning: FixedFormatter should only be used together with FixedLocator
  ax3.set_xticklabels(mean_arr_content.keys(), rotation=30, ha='right')
<ipython-input-135-5bcc6f65d002>:24: UserWarning: FixedFormatter should only be used together with FixedLocator
  ax4.set_xticklabels(mean_arr_wording.keys(), rotation=30, ha='right')
<ipython-input-135-5bcc6f65d002>:26: UserWarning: FixedFormatter should only be used together with FixedLocator
  ax1.set_xticklabels(perc75_class_content.keys(), rotation=30, ha='right')
<ipython-input-135-5bcc6f65d002>:28: UserWarning: FixedFormatter should only be used together with FixedLocator
  ax2.set_xticklabels(perc75_class_wording.keys(), rotation=30, ha='right')
```



## Section 6: Baseline Model (Q7, 10 points)

### Baseline model with no filtered data

I ran the following algorithms

1. Linear regression
2. Random Forest regression
3. Support vector Machines -I implemented but it gave poor results.
4. Gradient Boost Method

Approaches to increase performance

1. Add More Data. - can't
2. Treat Missing and Outlier Values. applied in some capacity
3. Feature Engineering. applied in some capacity
4. Feature Selection. applied
5. Multiple Algorithms.
6. Algorithm Tuning.
7. Ensemble Methods. applied
8. Cross Validation.

```
result_cols=['Model','Accuracy','Mean Squared Error']
final_result=pd.DataFrame(columns=result_cols)
final_result.set_index('Model')
```

```
def linear_reg_flow(x,y):
    x_train,x_test,y_train,y_test=train_test_split(x,y,test_size=0.2,random_state=0)
    # print("X_train:",x_train.shape)
    # print("X_test:",x_test.shape)
    # print("Y_train:",y_train.shape)
    # print("Y_test:",y_test.shape)
    linreg=LinearRegression()
    linreg.fit(x_train,y_train)
    y_pred=linreg.predict(x_test)
    # print(y_pred)
    Accuracy=r2_score(y_test,y_pred)*100
    # print(" Accuracy based on r2_score of the model is %.2f" %Accuracy)
    r_squared = linreg.score(x, y)
    #view R-squared value
    # print("Rsquared default of the model is %.2f" %r_squared)
    msqe=mean_squared_error(y_test, y_pred, squared=True)
    # print(" mean_squared_error of the model is %.2f" %msqe)
    result={}
```

```
    result['accuracy_r2_score']=Accuracy
    result['r-squared']=r_squared
    result['mean_squared_error']=msqe
    final_result.loc['Linear regression']=['Linear regression',Accuracy,msqe]
    # print(pd.DataFrame.from_dict(result))
    print("Linear Regression:: "+str(result))



def random_forest_flow(x,y):
    x_train,x_test,y_train,y_test=train_test_split(x,y,test_size=0.2,random_state=0)
    regressor = RandomForestRegressor(n_estimators=10,  random_state=0)
    # fit the regressor with x and y data
    regressor.fit(x_train,y_train.values.ravel())
    # test the output by changing values
    y_pred = regressor.predict(x_test)
    Accuracy=r2_score(y_test,y_pred)*100
    print(" Random Forest Regressor:: Accuracy based on r2_score of the model is %.2f" %Accuracy)
    msqe=mean_squared_error(y_test, y_pred, squared=True)
    print(" mean_squared_error of the model is %.2f" %msqe)
    final_result.loc['Random Forest']=['Random Forest',Accuracy,msqe]
    #view R-squared value
    # print("Rsquared default of the model is %.2f" %r_squared)
    # print(" Out of bag score of the model is %.2f" %regressor.oob_score_)
    # print(regressor.oob_score_)


def GBM_Regressor_flow(x,y):
    X_train, X_test, y_train, y_test = train_test_split(x,y, test_size=0.20, random_state=100)
    model = GradientBoostingRegressor(n_estimators=1000,criterion='friedman_mse', max_depth=8,min_samples_split=5,
                                  min_samples_leaf=5,max_features=3)

    y_train1=np.ravel(y_train)
    model.fit(X_train,y_train1)
    y_pred = model.predict(X_test)
    # print(y_pred.shape)
    Accuracy=r2_score(y_test,y_pred)*100
    print(" GBM_Regressor_flow:: Accuracy based on r2_score of the model is %.2f" %Accuracy)
    msqe=mean_squared_error(y_test, y_pred, squared=True)
    print(" mean_squared_error of the model is %.2f" %msqe)
    final_result.loc['GBM Regressor']=['GBM Regressor',Accuracy,msqe]
```

## Case 1 : Baseline Model

```
%timeit
# -----------------Base Model
# For Content
print("FOR content--------------")
x=dl1[['text_wc', 'prompt_text_wc','text_uwc', 'prompt_text_uwc', 'common_uq_wc_txt_ptext','common_uq_wc_ques', 'common_uq_wc_title']]
y=dl1[['content']]

linear_reg_flow(x,y)

print("FOR wording--------------")

y=dl1[['wording']]
linear_reg_flow(x,y)
```

```
FOR content--------------
Linear Regression:: {'accuracy_r2_score': 70.79528967410214, 'r-squared': 0.69436787625059, 'mean_squared_error': 0.5562391777204025}
FOR wording--------------
Linear Regression:: {'accuracy_r2_score': 52.554040106306, 'r-squared': 0.5386484559690312, 'mean_squared_error': 0.7027275760195345}
```

## Case 2 : Basleine Model with additonal parameters

```
# Linear regression with additional parameters
# ['text_wc', 'prompt_text_wc', 'text_uwc', 'prompt_text_uwc', 'common_uq_wc_txt_ptext',  'common_uq_wc_ques', 'common_uq_wc_title',
#  'text_ari_score', 'summary_len_percent', 'has_bad_words',  'text_similarity_score', 'grammar_correction_cnt' ]
dl_hot = pd.get_dummies(dl1, columns = ['has_bad_words'])
# print(dl_hot.columns)
print("FOR content--------------")
x=dl_hot[['text_wc', 'prompt_text_wc', 'text_uwc', 'prompt_text_uwc', 'common_uq_wc_txt_ptext',  'common_uq_wc_ques', 'common_uq_wc_title',
```

```
 'text_ari_score', 'summary_len_percent', 'has_bad_words_False', 'has_bad_words_True',  'text_similarity_score', 'grammar_correction_cnt' ]]
y=dl_hot[['content']]
linear_reg_flow(x,y)
print("FOR wording--------------")
x=dl_hot[['text_wc', 'prompt_text_wc', 'text_uwc', 'prompt_text_uwc', 'common_uq_wc_txt_ptext',  'common_uq_wc_ques', 'common_uq_wc_title',
 'text_ari_score', 'summary_len_percent', 'has_bad_words_False','has_bad_words_True', 'text_similarity_score', 'grammar_correction_cnt' ]]
y=dl_hot[['wording']]
linear_reg_flow(x,y)
```

```
FOR content--------------
Linear Regression:: {'accuracy_r2_score': 74.43729160949131, 'r-squared': 0.7403144235084986, 'mean_squared_error': 0.5204015192862845}
FOR wording--------------
Linear Regression:: {'accuracy_r2_score': 58.29792440297145, 'r-squared': 0.5982364401584903, 'mean_squared_error': 0.6588191480646984}
```

Based on these results we see that basic trimming of data didn't help much ,instead it seems it worsened the situation .. Why ? Because it created lot of empty spaces i.e. many words were further removed from the text making the fitting process of linear regression even more approximate and thereby further from accurate.

## Case 2 a Basleine Model with additonal parameters with specific parameters

Content oriented parameters 1. Text similarity 2. Has bad words 3. Unique words common

Wording oriented columns 1. Readability 2. Grammar 3. Summary size

However this too worsened the situation.

```
##
x=dl_hot[['text_wc', 'prompt_text_wc', 'text_uwc', 'prompt_text_uwc', 'common_uq_wc_txt_ptext',  'common_uq_wc_ques', 'common_uq_wc_title',
 'text_ari_score',  'has_bad_words_False', 'has_bad_words_True',  'text_similarity_score' ]]
y=dl_hot[['content']]
linear_reg_flow(x,y)
print("FOR wording--------------")
x=dl_hot[['text_wc', 'prompt_text_wc', 'text_uwc', 'prompt_text_uwc', 'common_uq_wc_txt_ptext',  'common_uq_wc_ques', 'common_uq_wc_title',
 'text_ari_score', 'summary_len_percent',  'grammar_correction_cnt' ]]
y=dl_hot[['wording']]
linear_reg_flow(x,y)
```

```
Linear Regression:: {'accuracy_r2_score': 73.80132691110097, 'r-squared': 0.7358709607034467, 'mean_squared_error': 0.5268351836456578}
FOR wording--------------
Linear Regression:: {'accuracy_r2_score': 55.65576630525918, 'r-squared': 0.5730887416440875, 'mean_squared_error': 0.679369357885844}
```

```
x=dl_hot[['text_wc', 'prompt_text_wc', 'text_uwc', 'prompt_text_uwc', 'common_uq_wc_txt_ptext',  'common_uq_wc_ques', 'common_uq_wc_title',
 'text_ari_score', 'summary_len_percent', 'has_bad_words_False', 'has_bad_words_True',  'text_similarity_score', 'grammar_correction_cnt' ]]
y=dl_hot[['content']]
linear_reg_flow(x,y)
print("FOR wording--------------")
x=dl_hot[['text_wc', 'prompt_text_wc', 'text_uwc', 'prompt_text_uwc', 'common_uq_wc_txt_ptext',  'common_uq_wc_ques', 'common_uq_wc_title',
 'text_ari_score', 'summary_len_percent', 'text_similarity_score', 'grammar_correction_cnt' ]]
y=dl_hot[['wording']]
linear_reg_flow(x,y)
```

```
Linear Regression:: {'accuracy_r2_score': 74.43729160949131, 'r-squared': 0.7403144235084986, 'mean_squared_error': 0.5204015192862845}
FOR wording--------------
Linear Regression:: {'accuracy_r2_score': 58.25104407193644, 'r-squared': 0.596964478243341, 'mean_squared_error': 0.6591893572687233}
```

Final Models

```
def Run_models(df,col_x):
  result_cols=['Model','Accuracy','Mean Squared Error']
  print("FOR content--------------")
  x=df[col_x]
  y=df[['content']]
  linear_reg_flow(x,y)
  random_forest_flow(x,y)
  GBM_Regressor_flow(x,y)
  print("final_result content--------------")
  print(final_result)
  print("FOR wording--------------")
  x=df[col_x]
```

```
    y=df[['wording']]
    linear_reg_flow(x,y)
    random_forest_flow(x,y)
    GBM_Regressor_flow(x,y)
    print(final_result)
    print("final_result wording--------------")
```

## Final Model comparison

preference- 1) Gradient Boost Regression 2) Random Forest Regression 3) Linear regression

Based on these test I found that Random Forest seems to be a better regression model for this problem and dataset given these features.

In terms of data i found that the maximum accuracy and minimum Mean Squ

```
# Case 1: primary features with basic data Basemodel
# print(dl1.columns)
print("=================================================================case 1 : Basemodel")
col_x=['text_wc', 'prompt_text_wc','text_uwc', 'prompt_text_uwc', 'common_uq_wc_txt_ptext','common_uq_wc_ques', 'common_uq_wc_title']
Run_models(dl1,col_x)

# Case 2: primary features with filtered data
print("=================================================================case 2 : Model 1 : primary features with filtered data")
Run_models(adv_dl1,col_x)

dl_hot = pd.get_dummies(dl1, columns = ['has_bad_words'])


# Case 3: All features with basic data ==> regression with additional parameters
# ['text_wc', 'prompt_text_wc', 'text_uwc', 'prompt_text_uwc', 'common_uq_wc_txt_ptext',  'common_uq_wc_ques', 'common_uq_wc_title',
#  'text_ari_score', 'summary_len_percent', 'has_bad_words',  'text_similarity_score', 'grammar_correction_cnt' ]
print("=================================================================case 3 Model 2 : All features with basic data")
col_x=['text_wc', 'prompt_text_wc', 'text_uwc', 'prompt_text_uwc', 'common_uq_wc_txt_ptext',  'common_uq_wc_ques', 'common_uq_wc_title',
 'text_ari_score', 'summary_len_percent', 'has_bad_words_False', 'has_bad_words_True',  'text_similarity_score', 'grammar_correction_cnt' ]
Run_models(dl_hot,col_x)

adv_dl2=adv_dl1
# adv_dl2[['text_ari_score', 'summary_len_percent', 'has_bad_words_False', 'has_bad_words_True',  'text_similarity_score', 'grammar_correctic
adv_dl2[['text_ari_score', 'summary_len_percent', 'has_bad_words_False', 'has_bad_words_True',  'text_similarity_score', 'grammar_correction_
# print(adv_dl2.columns)

# Case 4: All features with filtered data
print("=================================================================case 4  model 3 All features with filtered data")
Run_models(adv_dl2,col_x)
```

```
                              Model   Accuracy  Mean Squared Error
Linear regression  Linear regression  58.297924            0.434043
Random Forest          Random Forest  61.451075            0.401224
GBM Regressor          GBM Regressor  65.033336            0.383786
final_result wording--------------
case 4  model 3
FOR content--------------
Linear Regression:: {'accuracy_r2_score': 75.37660238166085, 'r-squared': 0.7502295435839056, 'mean_squared_error': 0.26086644746933
 Random Forest Regressor:: Accuracy based on r2_score of the model is 77.57
 mean_squared_error of the model is 0.24
 GBM_Regressor_flow:: Accuracy based on r2_score of the model is 80.14
 mean_squared_error of the model is 0.23
final_result content--------------
                              Model   Accuracy  Mean Squared Error
Linear regression  Linear regression  75.376602            0.260866
Random Forest          Random Forest  77.569250            0.237637
GBM Regressor          GBM Regressor  80.143975            0.226753
FOR wording--------------
Linear Regression:: {'accuracy_r2_score': 58.00951173470374, 'r-squared': 0.5995741218365745, 'mean_squared_error': 0.43704452055029
 Random Forest Regressor:: Accuracy based on r2_score of the model is 62.00
 mean_squared_error of the model is 0.40
 GBM_Regressor_flow:: Accuracy based on r2_score of the model is 64.95
 mean_squared_error of the model is 0.38
                              Model   Accuracy  Mean Squared Error
Linear regression  Linear regression  58.009512            0.437045
Random Forest          Random Forest  61.997744            0.395534
GBM Regressor          GBM Regressor  64.947592            0.384727
final_result wording--------------
```

Thus the best possible model to predict is GBM Regressor > Random Forest >Linear regression. We achieved best Root Mean Squared Error :

1. Content : 0.217141
2. Wording : 0.383786

for GBM Regressor with Accuracy in percentage:

1. Content : 80.985649
2. Wording : 65.033336

## Section 8: Kaggle Submission Screenshots (Q10, 5 points)

References


1. https://www.kaggle.com/competitions/commonlit-evaluate-student-summaries/data
2. https://www.geeksforgeeks.org/python-pandas-dataframe-dtypes/
3. https://saturncloud.io/blog/pandas-usecols-all-except-last-a-data-scientists-guide/#:~:text=What%20is%20usecols%3F,columns%20in%20the%20resulting%20DataFrame.
4. https://stackoverflow.com/questions/43297589/merge-two-data-frames-based-on-common-column-values-in-pandas
5. https://www3.cs.stonybrook.edu/~skiena/519/
6. https://saturncloud.io/blog/how-to-count-distinct-words-from-a-pandas-data-frame/
7. https://www.geeksforgeeks.org/how-to-apply-a-function-to-multiple-columns-in-pandas/
8. https://www.kdnuggets.com/2023/08/7-steps-mastering-data-cleaning-preprocessing-techniques.html
9. https://en.wikipedia.org/wiki/Automated_readability_index
10. https://en.wikipedia.org/wiki/Coleman%E2%80%93Liau_index
11. https://en.wikipedia.org/wiki/Flesch%E2%80%93Kincaid_readability_tests
12. https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.map.html
13. https://stackoverflow.com/questions/15228054/how-to-count-the-amount-of-sentences-in-a-paragraph-in-python
14. https://pypi.org/project/better-profanity/ but not working
15. https://pypi.org/project/profanity/
16. https://pypi.org/project/profanity-check/1.0.3/ not building
17. https://www.geeksforgeeks.org/python-program-to-calculate-the-number-of-digits-and-letters-in-a-string/
18. https://www.editage.com/blog/how-to-write-a-research-paper-summary/#:~:text=Structure%20and%20qualities%20of%20a%20good%20summary&text=It%20is%20typically%205%25%20to,or%20even%20just%20a%20sentence.
19. https://saturncloud.io/blog/how-to-split-a-column-of-tuples-in-pandas-dataframe/#:~:text=The%20easiest%20way%20to%20split,a%20value%20in%20the%20tuple.
20. https://www.geeksforgeeks.org/find-a-matrix-or-vector-norm-using-numpy/

21. https://github.com/SpencerPark/IJava
22. https://stackoverflow.com/questions/51287258/how-can-i-use-java-in-google-colab
23. https://stackoverflow.com/questions/45545110/make-pandas-dataframe-apply-use-all-cores
24. https://matplotlib.org/stable/gallery/subplots_axes_and_figures/subplots_demo.html
25. https://www.westga.edu/academics/research/vrc/assets/docs/scatterplots_and_correlation_notes.pdf
26. https://www.w3schools.com/python/matplotlib_scatter.asp
27. https://www.geeksforgeeks.org/how-to-plot-a-normal-distribution-with-matplotlib-in-python/
28. https://www.geeksforgeeks.org/python-pandas-dataframe-groupby/
29. https://www.geeksforgeeks.org/how-to-set-the-spacing-between-subplots-in-matplotlib-in-python/
30. https://www.geeksforgeeks.org/python-set-difference/
31. https://www.analyticsvidhya.com/blog/2022/03/multiple-linear-regression-using-python/
32. https://scikit-learn.org/stable/modules/generated/sklearn.metrics.mean_squared_error.html
33. https://www.analyticsvidhya.com/blog/2021/06/understanding-random-forest/
34. https://archive.ph/HiaWW
35. https://archive.ph/TfqyD
36. https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingRegressor.html
37. https://www.geeksforgeeks.org/random-forest-regression-in-python/
38. https://builtin.com/data-science/random-forest-python
39. https://towardsdatascience.com/feature-selection-using-random-forest-26d7b747597f
40. https://stackoverflow.com/questions/2866380/how-can-i-time-a-code-segment-for-testing-performance-with-pythons-timeit
41. https://www.analyticsvidhya.com/blog/2020/03/support-vector-regression-tutorial-for-machine-learning/
42. #BEGIN[ChatGPT BingChat]) Can you find some interesting points in the data if I give you the data?
43. #BEGIN[ChatGPT BingChat]) How can I send you the data?
44. #BEGIN[ChatGPT BingChat]) let me help you understand, they are summaries of prompts, based on it what can you infer?
45. #BEGIN[ChatGPT BingChat]) what models can I use to learn from this data and predict the content and wording scores

Backup code comment

Public Score:

Private Score:

Kaggle profile link: https://www.kaggle.com/philipanish011

Screenshot(s):