Chapter 5: Machine learning and NLP

5.1. Emotions classifier:

In this part we searched a lot in this part to find a suitable and applicable approach on our data, as we found

- DistilBert-base-uncased-emotion is a pretraining model it has a high accuracy 93% on emotion dataset from twitter but the emotions are limited they are just 7 which means that there are many emotions will not detect accurately

- Go-emotion Dataset from google research GoEmotions is designed to train neural networks to perform deep analysis of the tonality of texts. It is a labeled data 58,000 record marked up into 27 emotions divided into 12 positive and 11 negative and 4 ambiguous and 1 neutral but it will not be suitable for our unlabelled dataset of reviews.

- To enhance the model we fine-tuned a pre-trained BERT model (Bert-base-uncased) for emotion classification using the GoEmotions dataset will discuss the steps in implementation [5.1.4] but unfortunately the lack of resources prevent us from reaching a high accuracy we got an accuracy 52% , so we did not use it.

- A paper from IEEE considered about book literature emotion classifications[5], It has The emotions file we used as an emotion lexicon is limited it only contained adjectives and maps to the referred emotion they were almost 550 adjective but it has a large record to emotions 35, we use it and enhance this file to take more efficient results will discuss that at [5.1.4].

- The NRC Emotion Lexicon is designed to help in the automatic identification of emotions and sentiments expressed in text. It is a lexicon of words and their associated emotions and sentiments. We use it on our data for analysis, but we didn't recommend based on it

5.2. Dataset:

- We sourced our data from Kaggle, which includes a large dataset of 4 GB. To manage this size, we encountered challenges processing the entire dataset. Specifically, we worked with a CSV file containing approximately 3 million review records, linked to another CSV file of books through their titles [6].

- Handling the large dataset posed resource challenges when running it through pretrained models, consuming more resources than we could allocate. As a solution, we divided the data into manageable chunks for preprocessing, which will be detailed in the implementation section that follows.

5.3. Data Preparation:

1. One of the challenges we encountered is identifying reviews that may be considered noise, such as ambiguous statements like "I am sad that this book has ended," where the emotion expressed refers to something other than the book's content or impact. To address this, we conducted a preliminary study by sampling shuffled data and assessing the relevance of reviews. We evaluated how each review contributes to our model and whether it accurately reflects the book's themes and emotional content.

Relevance / chunks	Chunk1	Chunk2	Chunk3	Chunk4	Chunk5
	42/50	45/50	40/50	43/50	39/50

Hence, we made the decision to proceed with using the data, confident that we can address noise issues in the future through advanced techniques such as lexicons or other methods to ensure accuracy.

2. Before uploading the data, we undertook several steps to ensure its quality and completeness:

- We removed duplicate entries based on the 'Title' column to ensure each book is represented uniquely.

- Rows without a 'Title' were dropped to maintain data completeness.

- Specific books, such as "Its Only Art If Its Well Hung!", were removed for integrity reasons.

- Unnecessary columns ('ratingsCount', 'infoLink', 'previewLink') were deleted to streamline the dataset.

- Dates in the 'publishedDate' column were standardized to ensure consistency.

- Books lacking images were assigned a default URL, maintaining visual consistency across the dataset.

```
import pandas as pd
import csv

df = pd.read_csv('books_data.csv')

df = df.drop_duplicates(subset=['Title'], keep='first')
df = df.dropna(subset=['Title'])

index_to_drop = df[df['Title'] == "Its Only Art If Its Well Hung!"].index
df = df.drop(index_to_drop)

columns_to_delete = ['ratingsCount', 'infoLink', 'previewLink']
df = df.drop(columns=columns_to_delete, errors='ignore')

df['publishedDate'] = pd.to_datetime(df['publishedDate'], errors='coerce')

default_image_url = 'https://images.app.goo.gl/JT7nPtP5D9DxxUa19'
df['image'] = df['image'].fillna(default_image_url)

df.to_csv('Book.csv', index=False)
```

Figure 43- Data Processing

3. Several visualizations were created to analyze the book dataset, including the top ten books by reviews and the distribution of books across different categories.

4. We proceeded to prepare the reviews for the next steps, focusing on sentiment analysis and emotion classification, by following these steps:

- Convert all letters to lowercase.

- Replace punctuation with spaces.

- Eliminate duplicate letters used for exaggeration.

- Remove annotations enclosed within ** and () parentheses.

- Exclude quotes extracted from books.

- Eliminate URLs and email addresses.

- Remove new lines, tabs, and multiple spaces.

- Filter out stop words for clarity and focus.

```
merged_df['review/text'] = merged_df['review/text'].str.lower()
# Apply the function to the 'review' column
merged_df['review/text'] = merged_df['review/text'].apply(remove_punctuation)
# Remove duplicate letters used for exaggeration
merged_df['review/text'] = merged_df['review/text'].apply(lambda x: re.sub(r'(\w)\1{2,}', r'\1\1', x))
# Remove notes written between ** and ()
merged_df['review/text'] = merged_df['review/text'].apply(lambda x: re.sub(r'\*{2}.*?\*{2}|\(.*?\)', '', x))
# Remove quotes from the book
merged_df['review/text'] = merged_df['review/text'].apply(lambda x: re.sub(r'"(.*?)"', '', x))
# Remove URLs and emails
merged_df['review/text'] = merged_df['review/text'].apply(lambda x: re.sub(r'\b(?:https?://|www\.)\S+\b|[\w\.-]+@[\w\.-]+\.\w+', '', x))
# Remove new lines, tabs, multiple space characters
merged_df['review/text'] = merged_df['review/text'].apply(lambda x: re.sub(r'\s+', ' ', x))
```

```
def remove_stop_words(text):
    return ' '.join([word for word in text.split() if word not in final_stop_words])

lemmatizer = WordNetLemmatizer()
def lemma_text(text):
    tokens = word_tokenize(text)
    tokens = [lemmatizer.lemmatize(word) for word in tokens]
    return ' '.join(tokens)
def preprocess_text(text):
    text = remove_stop_words(text)
    text = lemma_text(text)
    return text
merged_df['review/text'] = Parallel(n_jobs=-1)(delayed(preprocess_text)(text) for text in merged_df['review/text'])
```

Figure 44- preprocessing reviews

5.3.1 Sentiment Analysis:

- We employed three approaches for sentiment analysis: TextBlob, NLTK analysis, and a pretrained Roberta model.

Table 2-sentiment analysis comparisons

Criteria	TextBlob	NLTK (VADER)	RoBERTa Pre-trained Model
Ease of Use	Very easy to use with simple API calls	Easy to use, slight learning curve for configuration	Requires more setup and transformer models and deep learning frameworks
Performance	Moderate, good for straightforward text	High for social media and short texts, moderate for others	Highest due to deep learning architecture capturing context and subtleties
Scalability	Scales well for small to medium-sized datasets	Scales well, suitable for real-time analysis of social media streams	Computationally intensive, requiring significant resources for large-scale datasets
Accuracy	Moderate accuracy, suitable for quick and simple sentiment analysis tasks	High accuracy for social media and short texts, moderate for others	Highest accuracy, effective for diverse and complex text due to deep contextual understanding
Customization	Limited, primarily lexicon-based	Some customizations possible through lexicon adjustments	Highly customizable through fine-tuning on specific datasets

- Despite the size of our dataset, we chose the Roberta pretrained model for its enhanced capability in understanding context and nuances of humor.

- When using Roberta, which has a 512-word limit per sentiment analysis instance, we encountered large reviews that exceeded this limit. To address this limitation, we experimented with three solutions:

1. To take only the first 512 word and this will indicate to rest of the review.

2. To divide the text into chunks and calculate the classification then take the maximum number of the chunks.

3. Calculating the average sentiment score across all text chunks.

- In the end, we decided to calculate the average sentiment score across all text chunks to mitigate the limitations of the Roberta model.the limitations of the Roberta model

```
from transformers import AutoTokenizer
from transformers import AutoModelForSequenceClassification
from scipy.special import softmax

MODEL = f"cardiffnlp/twitter-roberta-base-sentiment"
tokenizer = AutoTokenizer.from_pretrained(MODEL)
model = AutoModelForSequenceClassification.from_pretrained(MODEL)
def get_sentiment(text):
    if not text:
        return 'roberta_neutral', 0.0
    max_length = 512
    texts = [text[i:i+max_length] for i in range(0, len(text), max_length)]
    scores_sum = {
        'roberta_negative' : 0,
        'roberta_neutral' : 0,
        'roberta_positive' : 0
    }
    for txt in texts:
        encoded_text = tokenizer(txt, return_tensors='pt')
        output = model(**encoded_text)
        scores = output[0][0].detach().numpy()
        scores = softmax(scores)
        scores_dict = {
            'roberta_negative' : scores[0],
            'roberta_neutral' : scores[1],
            'roberta_positive' : scores[2]
        }
        for key in scores_sum.keys():
            scores_sum[key] += scores_dict[key]

    # Compute the average scores
    scores_avg = {key: val / len(texts) for key, val in scores_sum.items()}
    # Get the sentiment with the maximum average score
    overall_sentiment = max(scores_avg, key=scores_avg.get)

    return overall_sentiment, scores_avg[overall_sentiment]
data['sentiment'], data['sentiment_scores'] = zip(*data['preprocessed'].apply(get_sentiment))
```

Figure 45-Roberta sentiment analysis

- The majority of reviews to be positive ones to influence the visiting user to read the respective book.

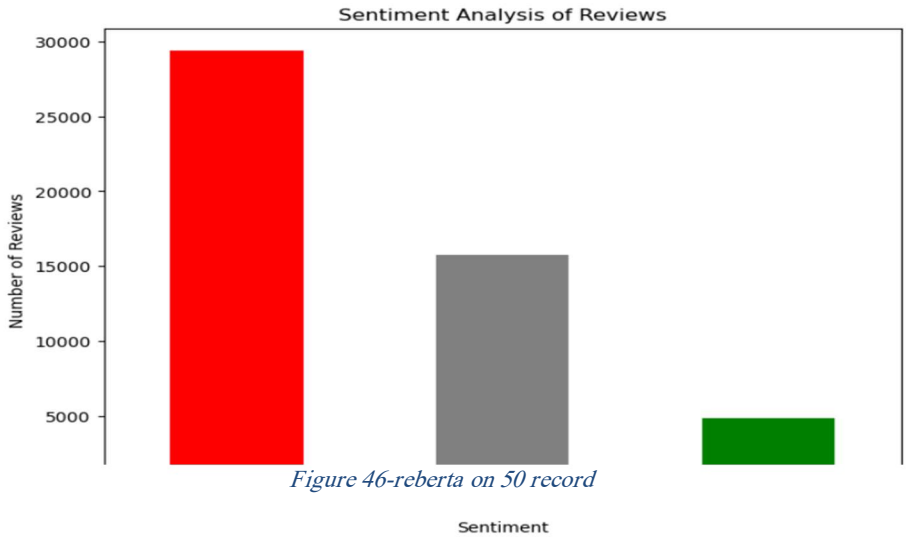

Figure 46-reberta on 50 record

5.4. Emotion classification models

- We made trials to fine-tuned a pre-trained BERT model (bert-base-uncased) for emotion classification using the GoEmotions dataset.

- The process involved several key steps:

1. Dataset Preparation: The GoEmotions dataset was loaded and tokenized using BERT's tokenizer, with padding and truncation applied to ensure uniform input lengths.

2. Model Configuration: A BERT model configured for sequence classification with 28 emotion labels was employed. The model's output was adapted to PyTorch tensors for efficient computation.

3. Training Setup: The training process included defining an AdamW optimizer and a linear learning rate scheduler. The model was trained over 20 epochs, with batch-wise forward and backward passes, followed by optimizer updates.

4. Evaluation and Metrics: The model's performance was evaluated using accuracy and F1 score metrics. These evaluations were conducted on a held-out test set, ensuring an unbiased assessment of the model's capabilities.

5. Model Deployment: The fine-tuned model and corresponding tokenizer were saved for deployment and further use in emotion classification tasks.

- The final model achieved an accuracy of 0.569% and a weighted F1 score of 0.56%, demonstrating its effectiveness in emotion classification on the GoEmotions dataset, it can be better accuracy, but our resources could not afford with more epochs.

```
Epoch 1 completed
Epoch 2 completed
Epoch 3 completed
Accuracy: 0.5697438732264603
F1 Score: 0.5639170209222097
```

Figure 47-finetuned bert with Go-emotions

- Second trail we go for emotion lexicon files we have nrc-emotion-lexicon-wordlevel-v0.92.txt and the emotion.txt of the paper we run both on sample of reviews and we manually grade the outputs true if the sentiment, emotion.txt, nrc.txt are relevance if not it would be false. It seems that the Nrc file has good results, but it has limitation on the number of emotions 7.

In the end, we took the file of the paper as it has 35 which is better, but it needs some enhancement as the text file has about 550 adjectives.

An emotion lexicon (emotions.txt) is used to map words to their respective emotions. This lexicon also includes mappings for negated words to their corresponding emotions.

- We expand this file using word net using the names and verbs that is corresponding to the adjective to enhance the emotion detection

```
import nltk
from nltk.corpus import wordnet
import ast
nltk.download('wordnet')
nltk.download('omw-1.4')
emotion_lexicon = {}
with open('emotions.txt') as f:
    for line in f:
        word, emotion = line.strip().split(': ')
        emotion_lexicon[word.strip("'")] = emotion.strip("',")
def find_synonyms(word):
    synonyms = set()
    for synset in wordnet.synsets(word):
        for lemma in synset.lemmas():
            synonyms.add(lemma.name())
    return synonyms
expanded_lexicon = emotion_lexicon.copy()
for word, emotion in emotion_lexicon.items():
    synonyms = find_synonyms(word)
    for synonym in synonyms:
        expanded_lexicon[synonym] = emotion
with open('expanded_emotions.txt', 'w') as f:
    for word, emotion in expanded_lexicon.items():
        f.write(f"'{word}': '{emotion}',\n")
```

Figure 48

6. Lemmatization: Reduce words to their base forms using part-of-speech tagging.

```
lemmatizer = WordNetLemmatizer()

# Function to get the part of speech tag for lemmatization
def get_wordnet_pos(word):
    tag = nltk.pos_tag([word])[0][1][0].upper()
    tag_dict = {"J": wordnet.ADJ, "N": wordnet.NOUN, "V": wordnet.VERB, "R": wordnet.ADV}
    return tag_dict.get(tag, wordnet.NOUN)

# Verify Lemmatization
words = ["enjoyed", "enjoying", "enjoys", "enjoyable"]
lemmatized_words = [lemmatizer.lemmatize(word, get_wordnet_pos(word)) for word in words]
print("Lemmatized Words:", lemmatized_words)
```

Figure 49

7. Negation Handling: Identify negation words and append "_NEG" to the subsequent word. To handle this Sentence: “I am not happy”

- Processed: happy_NEG

- Emotions: ['not_happy']

```python
def preprocess_text_with_negation_and_lemmatization(text):
    text = re.sub(r'[^\w\s]', '', text.lower())
    tokens = word_tokenize(text)
    stop_words = set(stopwords.words('english'))
    # Define the negative stop words that you want to exclude
    negative_stop_words = set(['not'])
    # Remove the negative stop words from the stop words set
    final_stop_words = stop_words - negative_stop_words
    tokens = [word for word in tokens if word not in final_stop_words]

    # Handle negations
    negation_words = set(['not', 'no', 'never', 'none'])
    negated = False
    processed_tokens = []

    for token in tokens:
        if token in negation_words:
            negated = True
            continue

        lemma = lemmatizer.lemmatize(token, get_wordnet_pos(token))

        if negated:
            processed_tokens.append(lemma + '_NEG')
            negated = False
        else:
            processed_tokens.append(lemma)

    return ' '.join(processed_tokens)
```

Figure 50

- If the token is negated, we make it to map to emotion negated and get the suitable emotion, like:

```
Sentence: I am not happy
Processed: not_happy
Emotions: ['sad']
```

Figure 51

- Then mapping to emotions negated emotions

```python
# Function to classify text based on the expanded lexicon
def classify_emotions_with_negation(text):
    emotions = []
    for word in text.split():
        if word in expanded_lexicon:
            emotions.append(expanded_lexicon[word])
        elif word.startswith('not_'):
            base_word = word[4:]
            if base_word in expanded_lexicon:
                original_emotion = expanded_lexicon[base_word]
                negated_emotion = f"not_{original_emotion}"
                if negated_emotion in negated_emotion_mapping:
                    emotions.append(negated_emotion_mapping[negated_emotion])
                else:
                    emotions.append(negated_emotion)
    return emotions
```

Figure 52

5.5. Content Based recommendation:

- Content-based recommendation systems suggest items to users based on the features of the items and a profile of the user's preferences. The main idea is to recommend items that are similar to those the user has liked in the past. In the context of books, the system might use features such as the book's title, author, category, and the emotions expressed in reviews.

-There are books have little number of reviews and that may make the recommendation based and not accurate so we consider books that has number of reviews.

- Steps:

1. Feature extraction (emotions, authors, category) and combine this features into a single string to each book

2. TF-IDF Vectorization Convert the combined feature strings into numerical vectors using TF-IDF (Term Frequency-Inverse Document Frequency). TF-IDF helps in identifying the importance of a word in a document relative to a collection of documents

3. User Profile Creation Create a profile for each user based on the books they have read and rated. The user profile is a weighted average of the TF-IDF vectors of the books they have read, with weights being the user's ratings for those books.

4. Recommendation Generation For a given user, calculate the similarity between the user's profile and the feature vectors of all books. Recommend the top 10 books with the highest similarity scores it is cold start for new users who aren't have user history and for users who has an user history will use it and consider the desired emotion

5.6. Calculate TF-IDF vector

1. TF-IDF is a statistical measure used to evaluate the importance of a word in a document

2. Relative to a collection of (corpus). The combined features are vectorized using TF-IDF

3. The TF-IDF matrix is converted to a DataFrame for better visualization.

4. Ratings are normalized using MinMaxScaler to scale them between 0 and 1.

5.7. Create user profile

1. Initialize an empty feature vector for the user.

2. For each book the user has read and rated:

3. Retrieve the book's feature vector.

4. Multiply the feature vector by the user's rating for that book (to give more weight to higher-rated books).

5. Add the weighted feature vector to the user's feature vector.

6. Average the user's feature vector by dividing by the total number of books the user has rated.

5.8. Applying similarity

- For existing users, the recommend_books_for_user function calculates the cosine similarity between the user profile vector and the feature vectors of all books.

- For new users or when a desired emotion is specified, the function calculates the cosine similarity between the desired emotion vector and the feature vectors of all books.

- Books are then recommended based on the highest similarity scores, combined with their ratings and review counts to filter and sort the recommendations

$$similarity(A,B) = \frac{A \cdot B}{\|A\| \times \|B\|} = \frac{\sum_{i=1}^{n} A_i \times B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \times \sqrt{\sum_{i=1}^{n} B_i^2}}$$

```python
# Function to recommend books for a user based on their profile and desired emotion
def recommend_books_for_user(user_profile, combined_features, df_books, desired_emotion, top_n=5):
    # Combine desired emotion with empty author and category for feature extraction
    desired_combined_features = ' '.join([desired_emotion, '', ''])

    # Vectorize the desired features
    desired_vector = tfidf.transform([desired_combined_features]).toarray()

    # Add zero for the normalized rating to match the dimension
    desired_vector = pd.DataFrame(desired_vector, columns=tfidf.get_feature_names_out())
    desired_vector['normalized_rating'] = 0

    # Calculate similarity between user profile and book features
    similarities = combined_features.apply(lambda x: linear_kernel([x], [user_profile])[0][0], axis=1)

    # Add similarity scores to the dataframe
    df_books['similarity'] = similarities

    # If the number of reviews is less than 5, set emotion similarity to 0
    df_books['emotion_similarity'] = df_books.apply(
        lambda row: 0 if row['num_reviews'] < 5 else linear_kernel([combined_features.iloc[row.name]], desired_vector)[0][0], axis=1
    )

    # Combine similarity scores with a weight factor (e.g., 0.5 for each)
    df_books['combined_similarity'] = 0.5 * df_books['similarity'] + 0.5 * df_books['emotion_similarity']

    # Filter and sort by combined similarity and rating
    recommended_books = df_books.sort_values(by=['combined_similarity', 'rating'], ascending=[False, False])

    return recommended_books[['title', 'author', 'category', 'rating', 'emotion', 'num_reviews']].head(top_n)
```

Figure 53