INFO SYS 723 Team 8 - Milestone 2

Team Name: Reading Lovers

Team Members: Yuchen Dai, Junchen Xia, Zijian Zhang, Joyce Xu

Part 1. Data Cleaning

As mentioned in Milestone 1, the Amazon book review dataset exhibits four data quality issues, which will be addressed systematically in sequence.

Screenshots of Datasets

	Title	description	authors	image	previewLink	publisher	publishedDate	infoLink	categories	ratingsCount
0	Its Only Art If Its Well Hung!	NaN	['Julie Strain']	http://books.google.com/books/content?id=DykPA...	http://books.google.nl/books?id=DykPAAAACAAJ&d...	NaN	1996	http://books.google.nl/books?id=DykPAAAACAAJ&d...	['Comics & Graphic Novels']	NaN

	Id	Title	Price	User_id	profileName	review/helpfulness	review/score	review/time	review/summary	review/text
0	1882931173	Its Only Art If Its Well Hung!	NaN	AVCGYZL8FQQTD	Jim of Oz "jim-of-oz"	7/7	4.0	940636800	Nice collection of Julie Strain images	This is only for Julie Strain fans. It's a col...

1.1 Irrelevant Features

After thorough inspection of the dataset, we have identified several features that are irrelevant or do not contribute to the potential analytics goals of our project. Therefore, we will proceed to remove them. Features to be removed includes:

- **id: The id of Book**
- **image: The url for book cover**
- **previewLink: Link to access this book on Google Books**
- **infoLink: Link to get more information about the book on Google Books**

Since there are 212,403 unique values for the column 'Title', and 221,998 unique values for the column 'Id', while the table 'book_data' and the table 'book_rating' are joined on the column 'Title'. To prevent possible confusion, we choose to exclude the feature 'Id'.

1.2 Data Integrity

In this section, we will address data integrity. As mentioned in Milestone1, for columns with varying degrees of missing values, we have deployed different processing methods.

1.2.1 Columns with Significant Missing Values

- **ratingsCount: The count of ratings for book** - Missing %: 77%
- **Price: The price of Book** – Missing %: 84%

Considering the substantial gap in data, we have opted to entirely remove this feature from our dataset.

1.2.2 Columns with Moderate Missing Values

- **profileName: Name of the user who rates the book** - Missing %: 19%

- **authors: Name of book authors** - Missing %: 15%
- **publisher: Name of the publisher** - Missing %: 36%
- **categories: Genres of books** - Missing %: 19%

The approach to managing columns with a moderate level of missing values is tailored according to the data type and the specific context of the features involved. For the above features in question and their respective proportion of missing values, we have chosen to replace the missing entries with **a new category labeled 'Unknown'** for these specific features.

- **description: Description of book** - Missing %: 32%

Recognizing its importance for text analytics due to the rich text information it provides, we have opted to impute these missing values with **a placeholder text labeled "Unknown"** for this feature.

- **publishedDate: The date of publish** - Missing %: 12%

We need to convert the feature 'pulishedDate' in following steps, thus we will process missing values in next part.

1.2.3 Columns with Minimal Missing Values

- **Title: Book Title**
- **review/summary: Summary of a text review**
- **review/text: Full text of a review**

These three features above exhibit missing values that constitute less than 0.0001% of the dataset's total size, we have chosen to remove the rows containing missing values for any such features entirely.

1.3 Data Consistency

- **publishedDate: The date of publish**

Notable inconsistencies are observed in the format of the publication dates within the "publishedDate" column, complicating its utilization. To address this, we intend to extract the year value from the "publishedDate" column and subsequently discard the original column. This streamlined approach will facilitate a more uniform and efficient analysis of publication dates.

After extracting the year value from the column 'publishedDate', we will replace missing values of the column 'publishedYear' with label 'Unknown'. Then we will replace missing values of the column 'publishedYear' with label 'Unknown'.

1.4 Data Readability

1.4.1 Time Conversion

- **review/time: time of given the review**

The column ‘review/time’ utilized the Unix Timestamp to represent the time of given reviews. For the consideration of readability in following analysis, we will convert values of ‘review/time’ to a standard date format.

```
book_rating['review/time'] = pd.to_datetime(book_rating['review/time'], unit = 's')
```

1.4.2 Removing Symbols

- **authors: Name of book authors**
- **categories: genres of books**

For readability and subsequent text analysis, we will remove unnecessary brackets and quotes from these two columns.

```
non_punct_author = []
non_punct_categories = []
for author, category in zip(book_data['authors'], book_data['categories']):
        letters_authors = re.sub(r'[\[\]\'"]', '', author)
        non_punct_author.append(''.join(letters_authors))
        letters_categories = re.sub(r'[\[\]\'"]', '', category)
        non_punct_categories.append(''.join(letters_categories))
book_data['authors'] = pd.Series(non_punct_author)
book_data['categories'] = pd.Series(non_punct_categories)
```

1.4.3 Reclassification of Categories

- **categories: genres of books**

This column presents 10,883 unique categories across 212,404 books, a level of granularity that may dilute meaningful analysis due to the excessive specificity of some categories.

Upon closer examination, we identified instances where categories could be generalized for greater relevance and coherence, such as reclassifying ‘New Zealand Fiction’ simply as ‘Fiction’. To address this issue, we are considering the application of rule-based approach for a more rationalized reclassification of these categories, aiming to streamline the dataset for more insightful analysis.

To investigate the distribution of categories, we will generate statistical charts for the top 50 most frequently appeared book categories in our dataset. By analyzing the top 50 categories, we aim to gain insight on reclassification of categories

Top 50 Most Frequently Appeared Book Categories

Rank	Categories	Counts	%	Rank	Categories	Counts	%	Rank	Categories	Counts	%
1	Unknown	41199	19.4	**18**	Medical	2079	0.98	**35**	Comics & Graphic Novels	1162	0.55
2	Fiction	23419	11.03	**19**	Art	2054	0.97	**36**	Nature	1146	0.54
3	Religion	9459	4.45	**20**	Body, Mind & Spirit	2049	0.96	**37**	Architecture	963	0.45
4	History	9330	4.39	**21**	Language Arts & Disciplines	2036	0.96	**38**	Transportation	921	0.43
5	Juvenile Fiction	6643	3.13	**22**	Health & Fitness	2030	0.96	**39**	Law	895	0.42
6	Biography & Autobiography	6324	2.98	**23**	Political Science	1955	0.92	**40**	Humor	799	0.38
7	Business & Economics	5625	2.65	**24**	Psychology	1913	0.9	**41**	Photography	756	0.36
8	Computers	4312	2.03	**25**	Philosophy	1864	0.88	**42**	American literature	737	0.35
9	Social Science	3834	1.81	**26**	Travel	1812	0.85	**43**	Antiques & Collectibles	697	0.33
10	Juvenile Nonfiction	3446	1.62	**27**	Technology & Engineering	1662	0.78	**44**	Drama	685	0.32
11	Science	2623	1.23	**28**	Self-Help	1519	0.72	**45**	Bible	667	0.31
12	Education	2611	1.23	**29**	Poetry	1500	0.71	**46**	Pets	630	0.3
13	Cooking	2445	1.15	**30**	Foreign Language Study	1404	0.66	**47**	Literary Collections	607	0.29
14	Sports & Recreation	2267	1.07	**31**	Crafts & Hobbies	1350	0.64	**48**	Young Adult Fiction	595	0.28
15	Family & Relationships	2178	1.03	**32**	Performing Arts	1305	0.61	**49**	Games	545	0.26
16	Literary Criticism	2147	1.01	**33**	Reference	1277	0.6	**50**	Gardening	532	0.25
17	Music	2106	0.99	**34**	Mathematics	1185	0.56	**Top 50**		**171,299**	**80.68**

Based on outputs above, we observe that the top 50 most frequently appeared categories represent 61.28% (excluding the 'Unknown' feature) of the total number of books in our dataset, while the remaining categories account for less than 40% of total number. Therefore, we will manually merge and reclassify these top 50 categories, then categorize the remaining niche categories based on keywords.

1.4.3.1 Merge and Reclassify Top 50 Categories

According to semantics, we will make following adjustments on the top 50 categories:

- categorize 'Young Adult Fiction' under 'Juvenile Fiction';
- categorize 'Bible' under 'Religion';
- categorize 'Nature' under 'Science';
- categorize 'Music', 'Performing Arts', 'Photography', 'Antiques & Collectibles' under 'Art';
- categorize 'Psychology', 'Political Science' under 'Social Science';
- merge 'Cooking', 'Crafts & Hobbies', 'Pets', 'Games', 'Gardening' into the new category 'Lifestyle & Leisure';
- merge 'Language Arts & Disciplines', 'Foreign Language Study' into the new category 'Language';
- merge 'Sports & Recreation', 'Health & Fitness' into the new category 'Sports & Fitness';
- merge 'Literary Criticism', 'Poetry', 'Drama', 'Literary Collections' into the new category 'Non-Fiction Literature';
- merge 'Family & Relationships', 'Body, Mind & Spirit' and 'Self-Help' into the new category 'Personal Development';
- merge 'Science' and 'Technology & Engineering' into the new category 'Science & Technology'.

1.4.3.2 Keyword Mapping

For those niche categories beyond the top 50 categories, we can deploy keyword mapping techniques for recategorization.

We will start with more specific keywords to ensure accuracy. For example, there's a category named 'Bible stories' in our dataset, we need to reclassify it under 'Religion' based on the keyword 'Bible', instead of classifying it based on the keyword 'stories'. Thus we will start from more specific keywords, such as Bible, Christian, Judaism and Buddhism.

```
cate.loc[cate['categories'].str.contains('bible|christian|judaism|muslim|religio|monastic|buddha|buddhism|buddhist|islam', case = False), 'categories'] = 'Religion'
```

Then we can use keywords related to professions (i.e. basketball player) to merge some niche categories into the category 'Biography & Autobiography'.

```
cate.loc[cate['categories'].str.contains('singer|actor|actress|musician|player|celebrit|scientist', case = False), 'categories'] = 'Biography & Autobiography'
```

Niche categories contain specific keywords such as decorations, ornaments, and furniture can be classified to the category 'Lifestyle & Leisure'. Categories contain keywords associated with sports can be categorize into the category 'Sports & Fitness'.

```
cate.loc[cate['categories'].str.contains('decorat|ornament|furniture|pets|garden|cook|baking|game|hobbies', case = False), 'categories'] = 'Lifestyle & Leisure'
cate.loc[cate['categories'].str.contains('fitness|sport|basketball|tennis|football|ballet|hockey|yoga|running|golf', case = False), 'categories'] = 'Sports & Fitness'
```

Next, we will perform mapping techniques for categories 'Language', 'Art', 'Mathematics', 'Business & Economics', 'History', 'Social Science', 'Computer' based on their respective keywords.

```
cate.loc[cate['categories'].str.contains('language|linguist|semantic', case = False), 'categories'] = 'Language'
cate.loc[cate['categories'].str.contains('painting|drawing|music|antique|collectible|stage|photograph|calligraphy', case =
False), 'categories'] = 'Art'
cate.loc[cate['categories'].str.contains('math|algebra|geometry|calculus', case = False), 'categories'] = 'Mathematics'
cate.loc[cate['categories'].str.contains('business|economic', case = False), 'categories'] = 'Business & Economics'
cate.loc[cate['categories'].str.contains('historic|history', case = False), 'categories'] = 'History'
cate.loc[cate['categories'].str.contains('politic|psych', case = False), 'categories'] = 'Social Science'
cate.loc[cate['categories'].str.contains('computer', case = False), 'categories'] = 'Computers'
```

As we have mapped niche keywords already, significantly lowering the possibility of misclassification, we will now proceed to map general keywords, such as fiction, science, and art.

Special consideration should be given to the following points:

- The string 'art' appears not only as an independent word but also within words, without any actual semantic meaning, we will use the expression r'\bart\b' to avoid mismatch.
- We will retain categories Fiction and Juvenile Fifction for subsequent analysis, therefore we will temporarily change the name of 'Juvenile Fiction' to 'Juvenile Fction' to prevent mapping errors.
- We will also temporarily change the name of the categories 'Social Science' and 'History' to prevent mapping errors.

```
# Art
cate.loc[cate['categories'].str.contains(r'\bart\b', case=False, na=False), 'categories'] = 'Art'

# Rename History
cate['categories'] = cate['categories'].replace('History', 'Histry')

# Juvenile Fiction
cate.loc[(cate['categories'].str.contains('juvenile', case=False)| \
          cate['categories'].str.contains('child', case=False)| \
          cate['categories'].str.contains('young', case=False) ) & \
         (cate['categories'].str.contains('novel', case=False) | \
          cate['categories'].str.contains(r'\bfiction\b', case=False) | \
          cate['categories'].str.contains(r'\bstor\b', case=False)), 'categories'] = 'Juvenile Fiction'
cate['categories'] = cate['categories'].replace('Juvenile Fiction', 'Juvenile Fction')

# Juvenile Non-Fiction
cate.loc[(cate['categories'].str.contains('juvenile', case=False)| \
          cate['categories'].str.contains('child', case=False)| \
          cate['categories'].str.contains('young', case=False) ) & \
         (cate['categories'].str.contains('nonfiction', case=False) | \
          cate['categories'].str.contains('non-fiction', case=False) | \
          cate['categories'].str.contains('poetry', case=False) | \
          cate['categories'].str.contains('litera', case=False) ), 'categories'] = 'Juvenile Nonfiction'
cate['categories'] = cate['categories'].replace('Juvenile Nonfiction', 'Juvenile Nonfction')

# Non-Fiction Literature
cate.loc[cate['categories'].str.contains('criticism|poetry|drama|litera', case = False), 'categories'] = 'Non-Fiction Literature'
cate['categories'] = cate['categories'].replace('Non-Fiction Literature', 'Non-Fction Literature')

# Fiction
cate.loc[cate['categories'].str.contains('fiction|stories|story|novel', case=False), 'categories'] = 'Fiction'

# Science
cate['categories'] = cate['categories'].replace('Social Science', 'Social Sci')
cate.loc[cate['categories'].str.contains('science', case = False), 'categories'] = 'Science & Technology'

# Convert name of categories to its original value
cate['categories'] = cate['categories'].replace('Juvenile Fction', 'Juvenile Fiction')
cate['categories'] = cate['categories'].replace('Juvenile Nonfction', 'Juvenile Nonfiction')
cate['categories'] = cate['categories'].replace('Non-Fction Literature', 'Non-Fiction Literature')
cate['categories'] = cate['categories'].replace('Social Sci', 'Social Science')
cate['categories'] = cate['categories'].replace('Histry', 'History')
```

1.4.3.3 Word Cloud

After reclassification, the top 20 most frequently appeared categories (excluding the column 'Other') represent 63.50% of the total number of books, indicating that the distribution of book categories is more concentrated now. Next we will generate a word cloud for categories outside the top 20 categories to check any missing keywords.

According to the word cloud, we will deploy keyword mapping technique again to reclassify categories containing these keywords into more general categories.

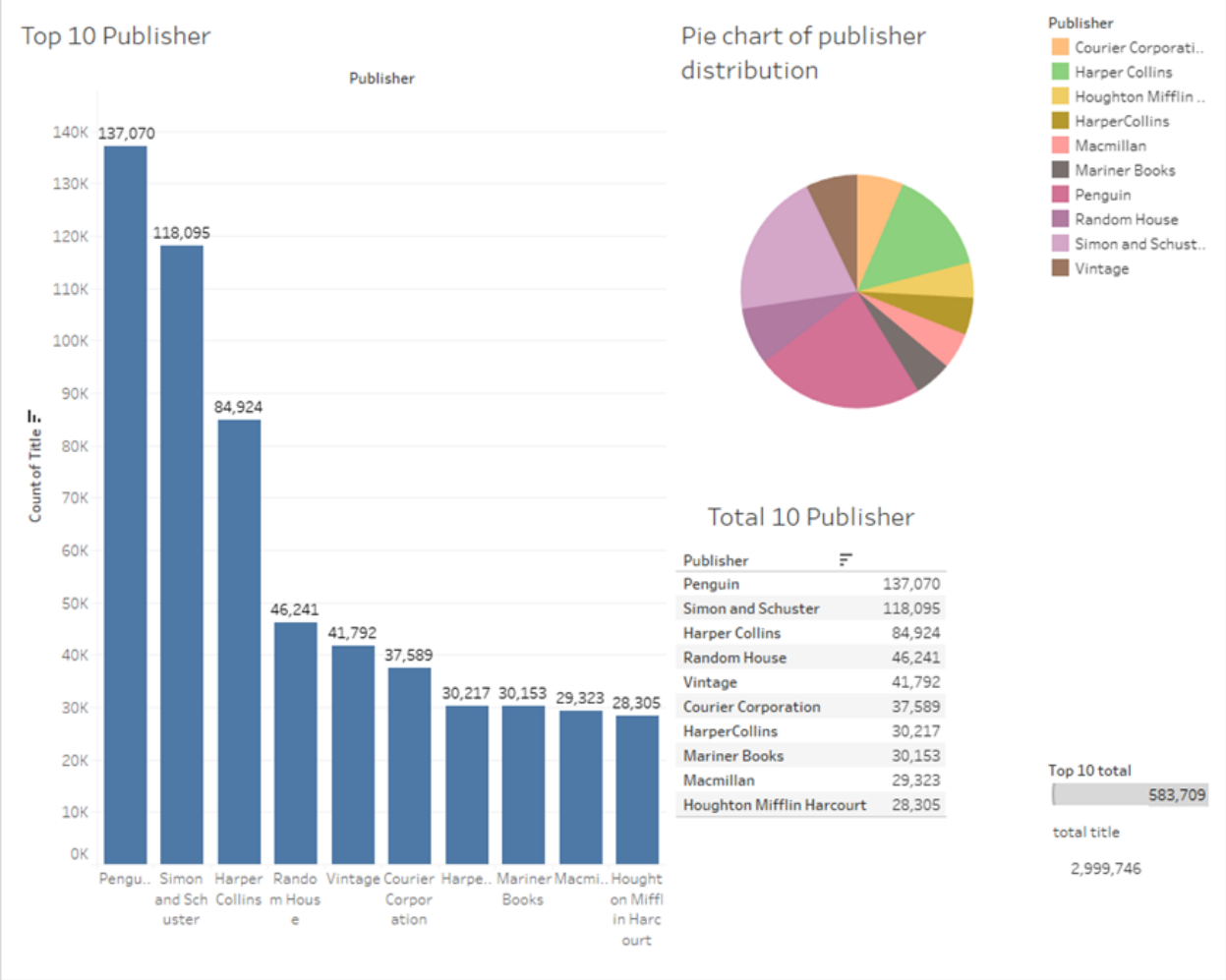
```
cate.loc[cate['categories'].str.contains('home|house', case = False), 'categories'] = 'Lifestyle & Leisure'
cate.loc[cate['categories'].str.contains('design', case = False), 'categories'] = 'Art'
cate.loc[cate['categories'].str.contains('automobile', case = False), 'categories'] = 'Science & Technology '
```

Part 2. EDA

2.1 Publisher Distribution:

2.1.1 Visualization

Our initial EDA will focus on examining the distribution of books across publishers. This investigation aims to uncover which publishers have the highest number of publications, shedding light on market dominance and competitive dynamics. Such insights will not only be valuable for benchmarking within the publishing industry but also assist new authors in making informed decisions about which publishers could offer the best platform for their work's success.

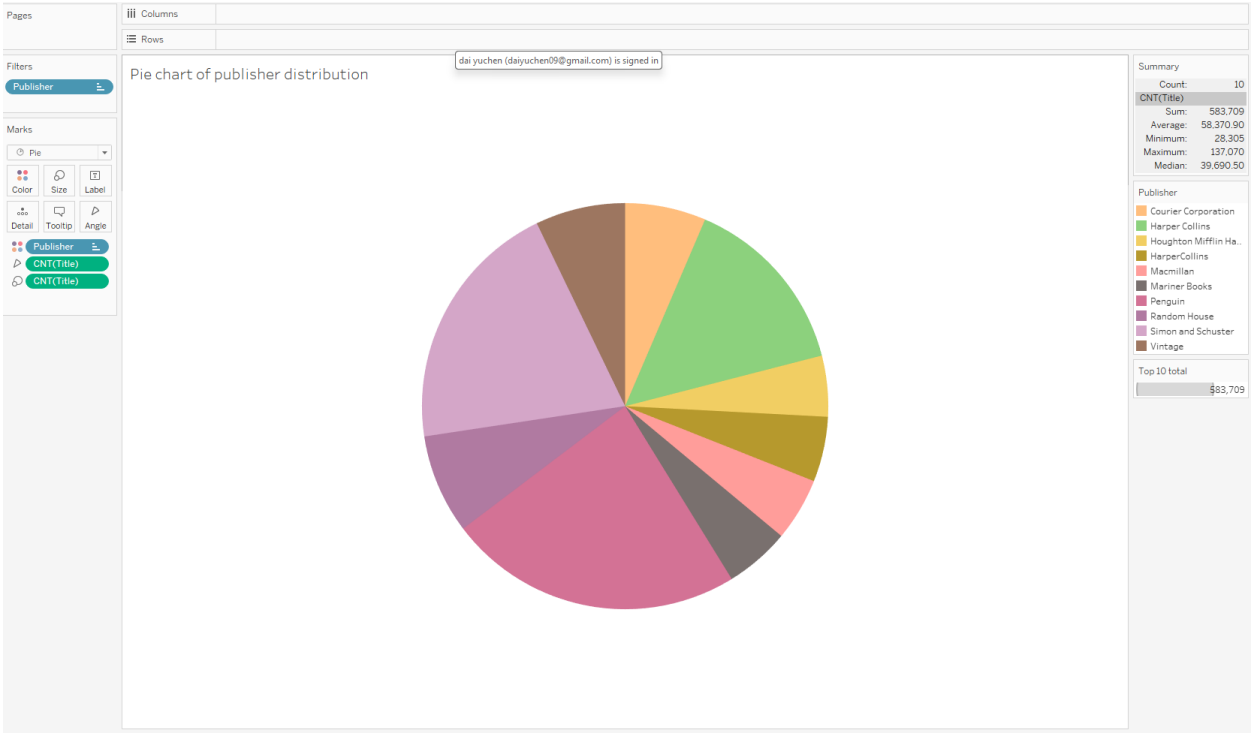


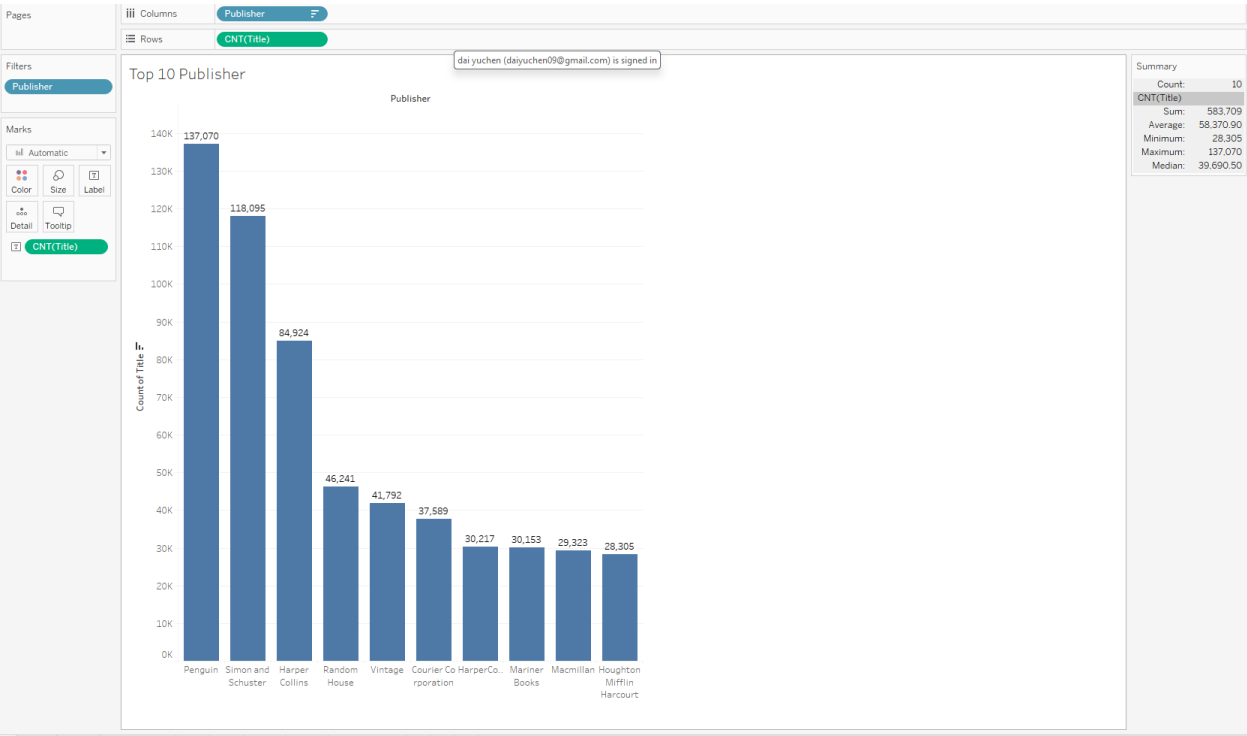
2.1.2 Insight

The bar chart and pie chart give a clear visual representation of the market share held by the top 10 publishers. Penguin and Simon & Schuster lead the pack with significantly higher publication counts, suggesting a potential oligopoly in the publishing industry. Together, the top

10 publishers account for 583,709 titles, which is a substantial portion of the total 2,999,746 titles in the dataset. This dominance indicates that these publishers might have a significant influence on what readers are exposed to, which could impact the diversity of available literature. The data also implies that these major publishers could be the most sought-after by authors seeking publication, given their market presence. This insight could be particularly useful for new authors in deciding which publishers to target, as well as for industry competitors in understanding market dynamics.

2.1.3 Key Steps

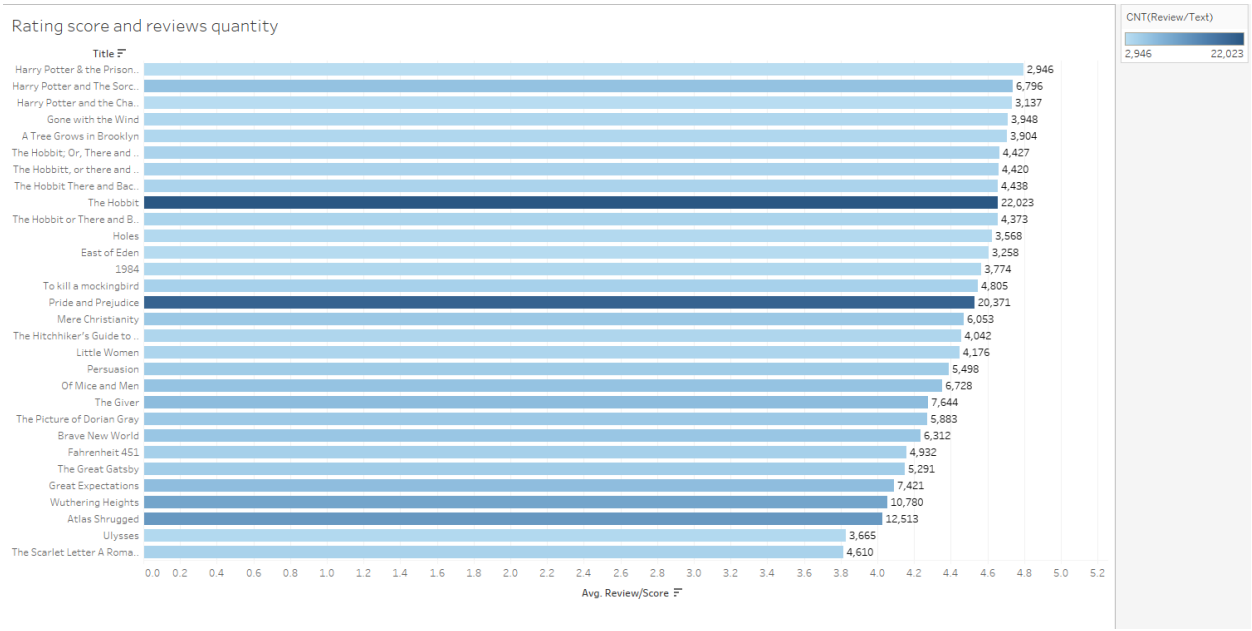




2.2 Ratings & Reviews Analysis

2.2.1 Visualization

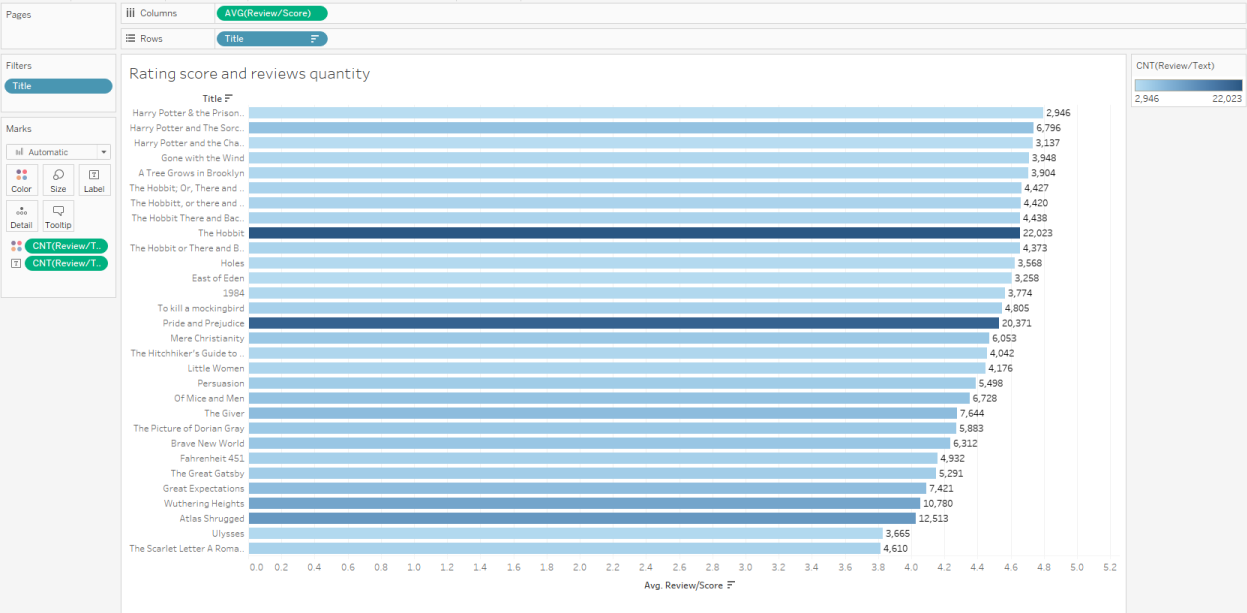
Next in our exploratory analysis, we will delve into the ratings associated with books that have garnered a high number of reviews. This inquiry is designed to identify books that excel in both review quantity and book ratings, providing a dual perspective on reader engagement and satisfaction. Understanding this dynamic is crucial for publishers and authors to fine-tune marketing strategies, as books demonstrating strong performance in these areas may warrant enhanced promotional activities, including advertising campaigns, strategic bookstore placements, or the release of special editions.



2.2.2 Insight

The bar chart showcases a clear trend: several standout books have both a high volume of reviews and high average ratings, indicating a robust reader response and satisfaction. This dual metric serves as a reliable indicator of both popularity and quality, which can be a powerful combination for marketing and sales strategies. Books like "Harry Potter & The Prisoner of Azkaban" and "Harry Potter & The Sorcerer's Stone" not only have a vast number of reviews but also maintain high ratings, underscoring their enduring appeal and the strong connection they have with their readers. Publishers and authors can leverage this information to target their promotional efforts more effectively. For example, books with high ratings and review counts could be prime candidates for advertising campaigns or special edition releases. Moreover, strategic placement in bookstores and online platforms could be optimized to capitalize on these titles' proven track record of reader engagement and satisfaction. This analysis underscores the potential of harnessing data to amplify success within the publishing industry.

2.2.3 Key Steps

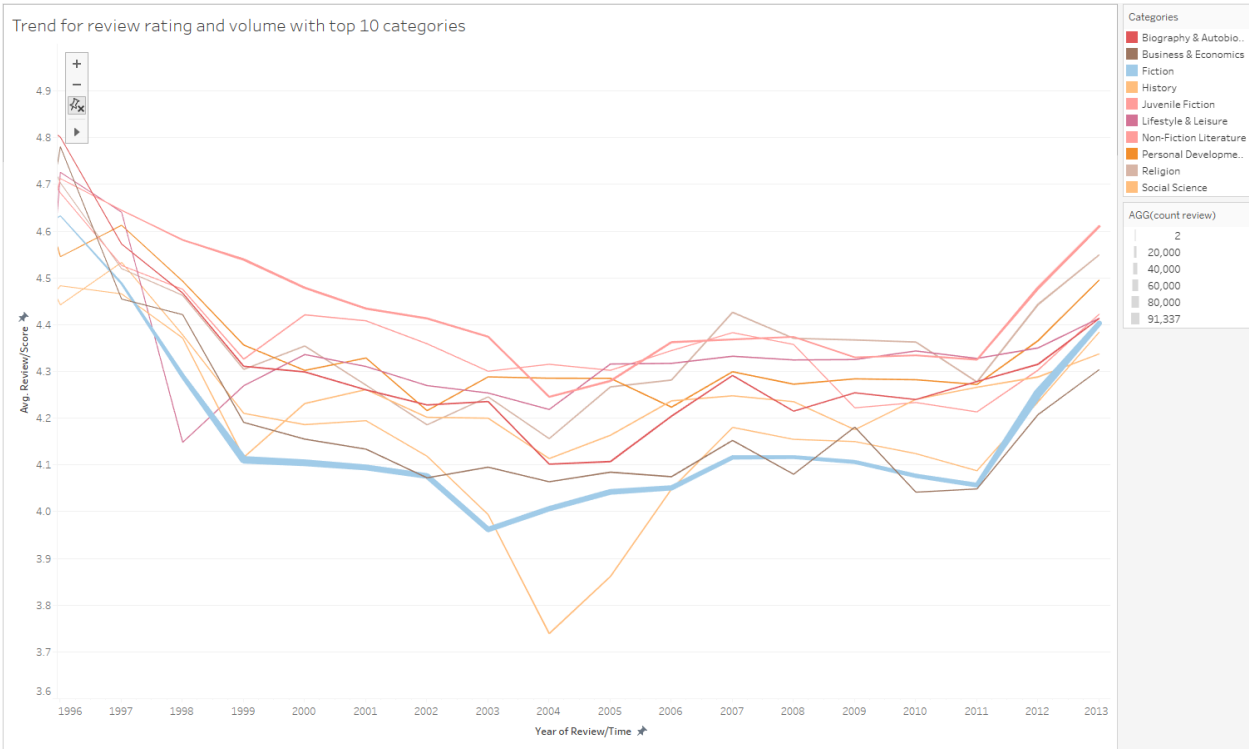


2.3 Trend Analysis Over Time:

2.3.1 Trend for review rating and volume with top 10 categories

2.3.1.1 Visualization

We will utilize the review/time feature to examine how book ratings and the volume of reviews have evolved over time. This analysis can reveal patterns, such as peaks in interest for certain types of books or shifts in reader engagement.



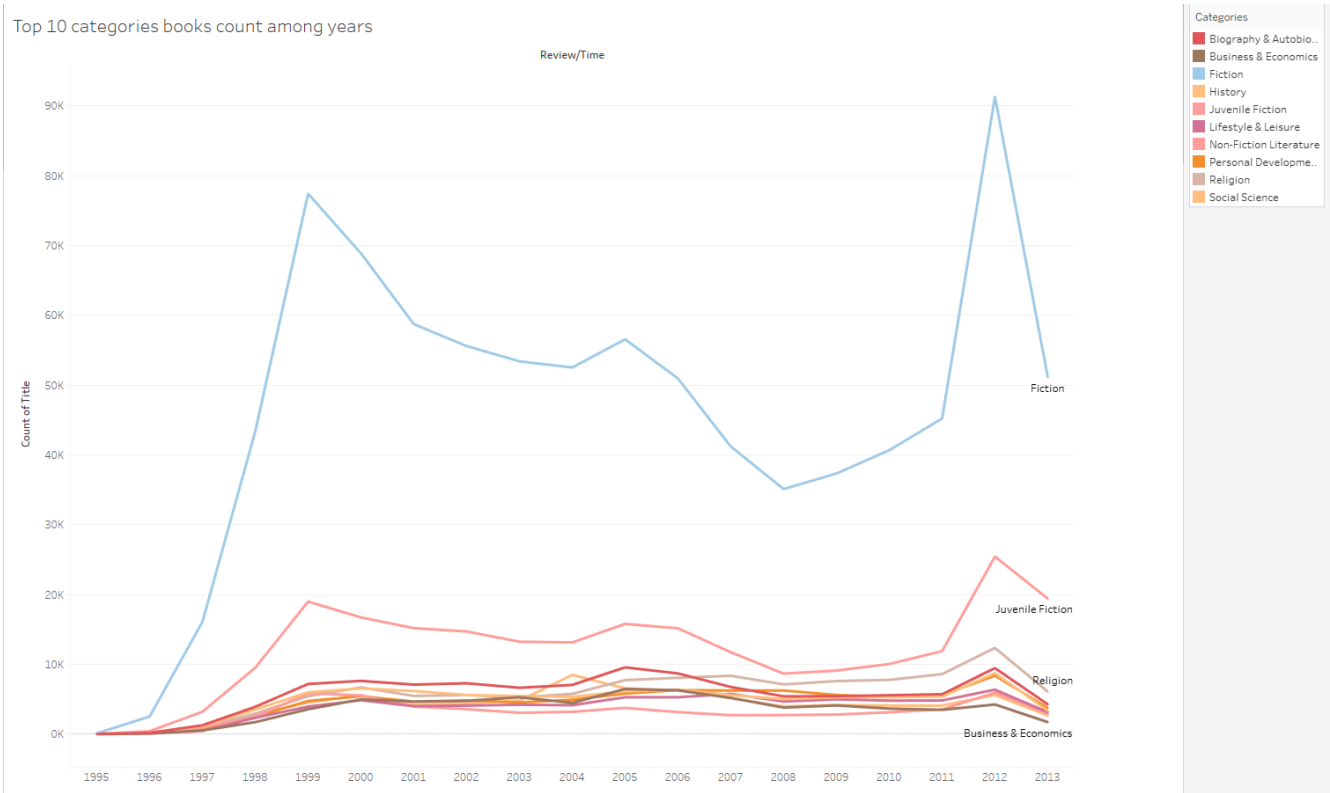
2.3.1.2 Insight

The line chart illustrates the trend in average book ratings across the top 10 categories from 1996 to 2013, revealing significant fluctuations in reader critiques over time. Categories such as 'Biography & Autobiography' and 'History' consistently show higher average ratings, which could indicate a more engaged and satisfied readership or a stable quality of publications in these genres. Some categories display marked volatility in ratings, notably 'Religion' and 'Social Science,' which could reflect the impact of influential new releases, shifts in societal interests, or varying critical standards. The volume of reviews, indicated by the thickness of the lines, suggests that 'Fiction' and 'Juvenile Fiction' not only attract a large number of reviews but also maintain relatively high ratings over the years, underscoring their popularity and potential market impact. The intersection of high review counts with strong ratings in these genres may point to a loyal and active reader base, offering a strategic advantage for publishers and authors when allocating resources and planning marketing campaigns. This EDA can serve as a strategic tool for industry stakeholders to identify sustainable trends and adjust to the evolving literary market.

2.3.2 Top 10 categories books count across year

2.3.2.1 Visualization

The goal of this EDA is to examine the volume of books published across the top 10 literary categories over a set period. By tracking the number of books reviewed each year, we can discern trends in publishing volume, detect burgeoning interest in specific genres, and identify potential saturation in the market. This analysis could reveal the ebb and flow of genre popularity, the impact of cultural and social phenomena on publishing trends, and the potential influence of significant literary awards or industry milestones.



2.3.2.2 Insight

The above graph depicts the count of book titles reviewed per year across the top 10 literary categories, highlighting significant variances over time. A striking feature is the dramatic spike in the 'Fiction' category, which towers over the others around the year 2000, suggesting an event or series of events that caused a surge in publishing or a heightened interest in fiction during that time. Following this peak, there's a notable dip, which could reflect a normalization after a trend or a market adjustment. The 'Juvenile Fiction' category shows a steady increase in the latter years, indicating growing popularity or increased focus on youth literature. The 'Religion' category and 'Business & Economics' display a modest yet steady presence over the years, suggesting a consistent production and review rate. This graph provides valuable insights into the publishing trends, indicating periods of heightened interest that may correlate with cultural shifts, economic conditions, or significant publishing phenomena. Such data can aid publishers and authors in understanding the historical market performance of different genres and adjusting their strategic planning accordingly.

Part 3. Text Classification

3.1 Overview

This part of our project embarks on a sophisticated classification endeavor, aimed at automatically sorting book reviews into three sentiment-based categories: Positive, Neutral, and Negative. Utilizing a suite of advanced natural language processing (NLP) techniques alongside a diverse array of machine learning models—namely, **Naïve Bayes, Logistic Regression, Random Forest,** and a cutting-edge **BERT-LSTM Neural Network**—we endeavor to navigate the intricate layers of meaning within textual feedback.

3.2 General Data Preparation

3.2.1 Label Transformation

For sentiment categorization, reviews are classified based on their star ratings: **4-5 stars as Positive, 3 stars as Neutral, and 1-2 stars as Negative.** We then employ **label encoding** (compared to One-Hot encoding) to facilitate our multi-class classification task, assigning 0 to Negative, 1 to Neutral, and 2 to Positive. This numeric transformation simplifies the model's understanding of sentiment labels and ensures compatibility with our selected models, enhancing processing efficiency without compromising the nuances of sentiment analysis.

3.2.2 Sampling & Balancing

To manage computational demands and ensure data balance, we randomly **sample 10,000 observations** from **each sentiment class**. This balanced sampling approach addresses computing resource limitations and helps avoid model bias by providing equal representation of each sentiment class, crucial for achieving accurate and generalizable classification results.

3.2.3 Train-Test Split

We adhere to a standard practice of partitioning our dataset into a **75%-25% split** for the training and test sets, respectively. For models undergoing a **tuning** process, additional dataset splits or techniques (i.e. cross-validation) may be necessary. These specific split configurations will be elaborated upon at the individual model level to accommodate their unique validation requirements.

3.2.4 Text Preprocessing & Representation

Different models necessitate distinct approaches to text preprocessing and representation due to their unique requirements. Therefore, we customize these steps for each model, **detailing specific techniques at the individual model level**. This tailored approach ensures optimal

model performance by aligning preprocessing methods with each model's capabilities, from simpler algorithms like Naïve Bayes to complex neural networks like BERT-LSTM.

3.3 Model 1: Naïve Bayes Regression

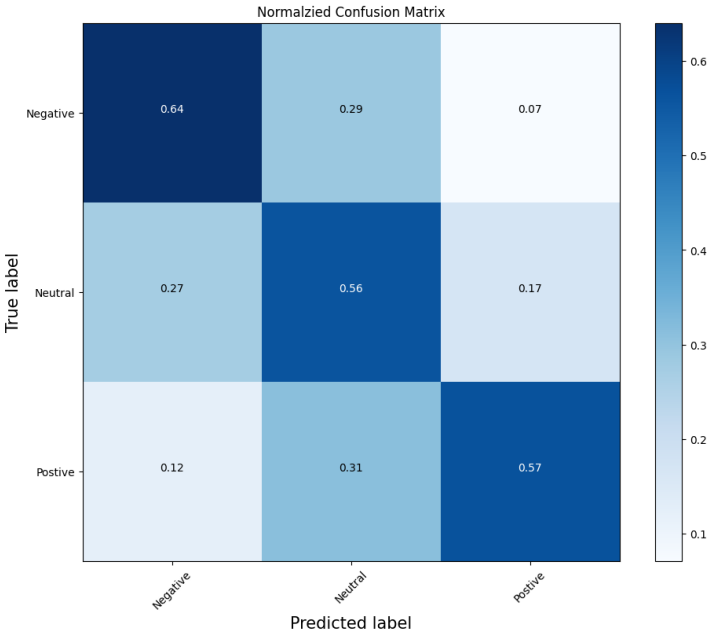
We start with basic text representation using bag of words and Naive Bayes. We set to 1 gram.

```
vect = CountVectorizer(preprocessor=clean
X_train_dtm = vect.fit_transform(X_train)
X_test_dtm = vect.transform(X_test)
print(X_train_dtm.shape, X_test_dtm.shape
```

```
(22500, 111207) (7500, 111207)
```

```
nb = MultinomialNB()
%time nb.fit(X_train_dtm, y_train)
y_pred_class = nb.predict(X_test_dtm)
```

```
Accuracy:  0.5890666666666666
Recall (macro): 0.5894975726767336
F1 score (macro): 0.5920621348499608
```



The performance is not that ideal.

3.4 Model 2: Random Forest Model

3.4.1 Basic Model

Then, we tried basic Random Forest model.

```
Accuracy:  0.6126666666666667
Recall (macro): 0.6124445356343585
F1 score (macro): 0.6101321066028204
```

It improved a little bit. Therefore, we used GridSearch to fine tune the hyperparameters.

3.4.2 Fine-Tuning the Random Forest Classifier

We use hyperparameter tuning to optimize Random Forest model for superior performance. We use GridSearchCV in **Python**.

```
BOW_vec1 = CountVectorizer(preprocessor=clean, ngram_range = (1, 1))
X_train_BOW = BOW_vec1.fit_transform(X_train)
X_test_BOW = BOW_vec1.transform(X_test)
print(X_train_BOW.shape, X_test_BOW.shape)
```

```
(22500, 111207) (7500, 111207)
```

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split, GridSearchCV
```

```
param_grid = {
    'n_estimators': [50, 100, 150],
    'max_depth': [None, 10, 20],
    'min_samples_split': [2, 5, 10],
    'max_features': ['sqrt', 'log2']
}
```

We specify a range of potential values for various hyperparameters to create an exploration grid, as depicted above. In this case, the grid includes different permutations of the number of estimators (**n_estimators**), the maximum depth of the trees (**max_depth**), the minimum number of samples required to split an internal node (**min_samples_split**), and the number of features to consider when looking for the best split (**max_features**).

And then, we created the Random Forest model with a fixed random_state for reproducibility. We initialize the GridSearchCV object to conduct a comprehensive search over the predetermined hyperparameter grid. The 'estimator' is our chosen model for optimization, the 'param_grid' denotes our specified grid for hyperparameter exploration, 'cv' represents the count of cross-validation folds to be used, and 'n_jobs' defines the quantity of CPU cores employed for concurrent computations, with '-1' indicating the utilization of all cores accessible.

```
rf=RandomForestClassifier(random_state = 91)
```

```
# Initialize GridSearchCV
grid_search = GridSearchCV(estimator=rf, param_grid=param_grid, cv=5, n_jobs=-1)
```

```
grid_search.fit(X_train_BOW,y_train)
```

```
▸ GridSearchCV
▸ estimator: RandomForestClassifier
  ▸ RandomForestClassifier
```

```
# Get the best parameters and the best estimator
best_params = grid_search.best_params_
best_rf = grid_search.best_estimator_
```

After that, we fit the model. GridSearchCV executes cross-validation with the designated fold count, systematically evaluating each combination of hyperparameters. After obtaining the best hyperparameter configuration, we use the tuned model to make predictions and calculate F1 score.

```
# Make predictions
y_pred = best_rf.predict(X_test_BOW)
```

```
print("Accuracy: ", metrics.accuracy_score(y_test, y_pred))
print("Recall (macro):",metrics.recall_score(y_test, y_pred, average='macro'))
print("F1 score (macro):",metrics.f1_score(y_test, y_pred, average='macro'))
cnf_matrix = confusion_matrix(y_test, y_pred)
plt.figure(figsize=(10,8))
plot_confusion_matrix(cnf_matrix, classes=['Negative', 'Neutral', 'Postive'],normalize=True,
                      title='Normalzied Confusion Matrix')
```

```
Accuracy:  0.6281333333333333
Recall (macro): 0.6278786012280948
F1 score (macro): 0.6256764383561237
```

It improved a little.

3.5 Model 3: Logistic Regression

3.5.1 Text Preprocessing

In our preprocessing workflow, after scanning through samples of the review texts, we opt to implement a series of standard steps to clean and prepare the text data. These steps include **removing digits and punctuation, removing stop words, converting text to lowercase, and tokenization**. The choice of normalization technique is tailored to align with our chosen text representation method. For **basic vectorization** tasks, we employ **stemming**, as it suffices to capture the broader meaning of words without necessitating exact dictionary forms. Conversely, when preparing data for **word embedding** techniques, we opt for **lemmatization**. This choice is driven by the need for precise dictionary forms of words to ensure accurate matching with the pre-defined vocabulary of the embeddings. Example preprocessing codes for the two cases are given below:

- For Basic Vectorization (Stemming):

```
def clean(doc):
    # Remove punctuation and digits, and lowercase the text
    doc = "".join([char.lower() for char in doc if char not in string.punctuation and not char.isdigit()])
    # Initialize the stemmer
    stemmer = PorterStemmer()
    # Tokenize, remove stopwords, and stem the words
    doc = " ".join([stemmer.stem(token) for token in doc.split() if token not in stopwords])
    return doc
```

- For Word Embedding (Lemmatization):

```
def preprocess_corpus(texts):
    def remove_stops_digits(tokens):
        #Nested function that lowercases, removes stopwords and digits from a list of tokens
        tokens_new = [wn.lemmatize(word) for word in tokens ]
        return [token.lower() for token in tokens_new if token.lower() not in mystopwords and not token.isdigit()
                and token not in punctuation]
    #This return statement below uses the above function to process twitter tokenizer output further.
    return [remove_stops_digits(word_tokenize(text)) for text in texts]
```

3.5.2 Text Representation

To explore the influence of text representation on model performance, we've selected an array of text representation strategies, combining various basic vectorization techniques (**BOW and TF-IDF**) with a range of n-gram configurations (**1 - gram, 1+2 - grams, and 1+2+3 - grams**). Additionally, we incorporate **word embeddings with 300 dimensions** to further enrich our analysis. This diverse suite of text representations aims to uncover how each method affects the effectiveness of our fundamental machine learning approach, specifically our basic logistic regression model. The following section provides a glimpse into the implementation of one set of basic vectorization representation, 300-dimension word embedding, and the logistic regression model employed in this study:

- **TF-IDF 1+2 - grams Vectorization:**

```
# Preprocess and Vectorize train and test data using TF-IDF Vectorization
tfidf_vec2 = TfidfVectorizer(preprocessor=clean, ngram_range = (1, 2)) # instantiate a vectoriezer
X_train_tfidf = tfidf_vec2.fit_transform(X_train) # use it to extract features from training data
# transform testing data (using training data's features)
X_test_tfidf = tfidf_vec2.transform(X_test)
print(X_train_tfidf.shape, X_test_tfidf.shape)
```

- **Creating a 300-dimension Word Embedding using "glove-wiki-gigaword-300":**

```
# Creating a feature vector by averaging all embeddings for all sentences
def embedding_feats(list_of_lists):
    DIMENSION = 300
    zero_vector = np.zeros(DIMENSION)
    feats = []
    for tokens in list_of_lists:
        feat_for_this =  np.zeros(DIMENSION)
        count_for_this = 0 + 1e-5 # to avoid divide-by-zero
        for token in tokens:
            if token in w2v_model:
                feat_for_this += w2v_model[token]
                count_for_this +=1
        if(count_for_this!=0):
            feats.append(feat_for_this/count_for_this)
        else:
            feats.append(zero_vector)
    return feats


train_vectors = embedding_feats(X)
```

- **Initiating a Logistic Regression Model using "sklearn" package:**

```
from sklearn.linear_model import LogisticRegression # import

logreg = LogisticRegression(class_weight="balanced") # instantiate a logistic regression model
logreg.fit(X_train_tfidf, y_train) # fit the model with training data

# Make predictions on test data
y_pred_class = logreg.predict(X_test_tfidf)
```

To assess the efficacy of our varied text representation techniques, we conducted a thorough evaluation of their performance on the test dataset. Our goal was to determine which method of text representation delivers the most favorable outcomes when applied to a basic logistic regression model. The outcomes of this evaluation, based on three metrics (**Accuracy, Recall [macro], & F-1 Score [macro]**), are detailed below:

	TF-IDF 1 gram	BOW 1 gram	TF-IDF 1+2 gram	BOW 1+2 gram	TF-IDF 1+2+3 gram	BOW 1+2+3 gram	Word Embd. 300-Dim.
Acc.	0.6396	0.6168	0.6479	0.6492	0.6417	0.6495	0.6006
Recall	0.6390	0.6163	0.6475	0.6486	0.6416	0.6488	0.6001
F-1 Score	0.6378	0.6153	0.6482	0.6465	0.6439	0.6460	0.5992

Our analysis reveals that **basic vectorization methods outperform word embeddings** in terms of efficacy. Specifically, TF-IDF marginally surpasses BOW when limited to single terms. The inclusion of 2-grams enhances performance for both vectorization techniques, rendering them similarly effective, whereas the addition of 3-grams offers negligible further improvement. Considering F-1 scores and with an eye on computational efficiency, we choose the **TF-IDF 1+2 grams** approach for our subsequent model development phase.

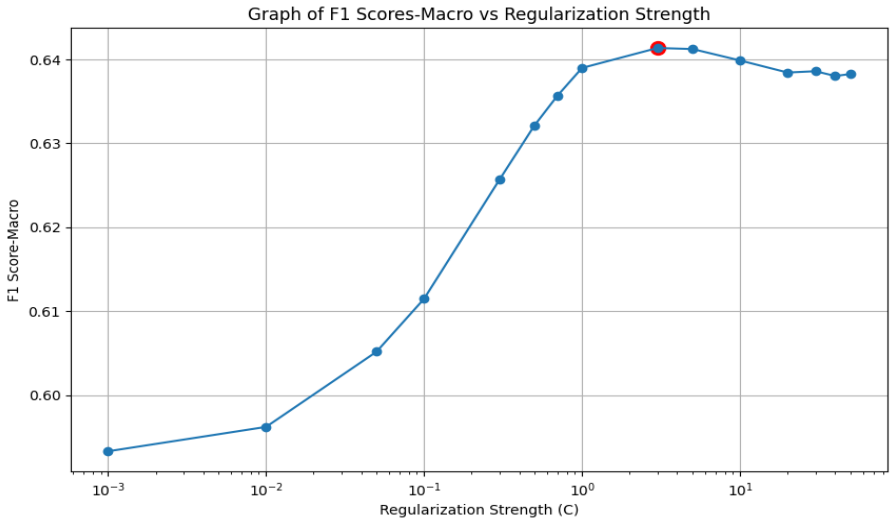
3.5.3 Parameters Tuning - L2 Regularization Term

After identifying the optimal text representation for our task, we further refined our logistic regression model by introducing an **L2 Ridge regularization term**. This adjustment aimed to explore potential performance improvements. To optimize the **regularization strength (C)**, we conducted a **5-fold cross-validation**, evaluating various C values to determine the one yielding the highest validation scores. The configuration for this validation process and the corresponding performance outcomes are detailed below:

- **5-Fold Cross-Validation setup:**

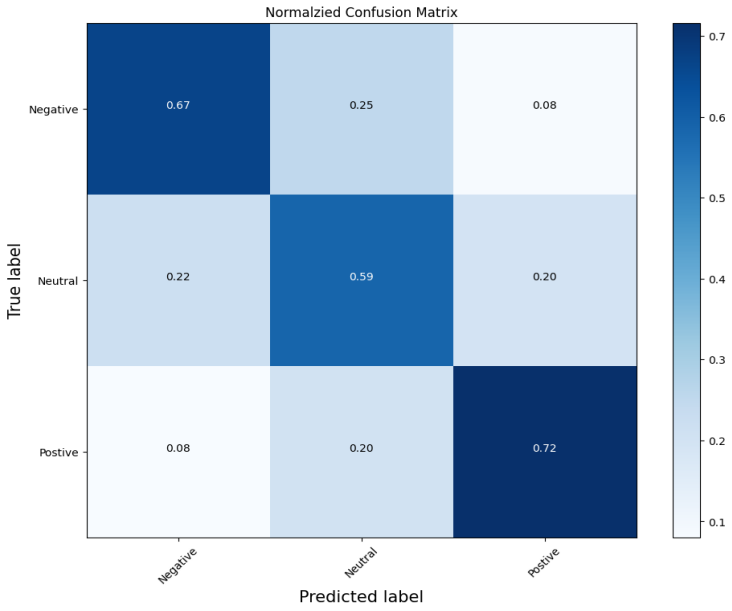
```
C = [50, 40, 30, 20, 10, 5, 3, 1, 0.7, 0.5, 0.3, .1, 0.05,.01,.001]
mean_scores = []
for c in C:
  lassologcv_model = LogisticRegression(penalty='l2',C=c,class_weight = 'balanced',solver='saga',max_iter=2000)
  scores = cross_val_score(lassologcv_model, X_train_tfidf, y_train, cv=5, scoring="f1_macro")
  mean_score = np.mean(scores)
  mean_scores.append(mean_score)

print(mean_scores)
```

- **Regularization Strength vs. Validation Performance (F-1 Score [Macro]) - Result:**



Within our selected range, **a C value of 3** produced the best validation performance for our L2-regularized logistic regression model. Further tuning around C=3, with values [4.5, 4, 3.5, 3, 2.5, 2, 1.5], confirmed that **C=3 remains optimal.** Consequently, we chose **C=3 as our definitive L2 regularization strength**. After retraining on the entire training set with C=3, we evaluated the model on the test set. The performance metrics of our finalized logistic regression model with L2 regularization are presented below:

- **Performance Metrics:**
  - Accuracy: 0.6566
  - Recall (macro): 0.6563
  - F1 score (macro): 0.6571
- **Confusion Matrix:**



The inclusion of a finely-tuned L2 regularization term has resulted in a **discernible improvement** across all three performance metrics for our logistic regression model, compared to its counterpart without L2 regularization. This enhancement underscores the value of regularization in optimizing model performance.

3.6 Model 4: BRET-LSTM Neural Network

3.6.1 Specific Data Preparation

3.6.1.1 Text Preprocessing

The BERT model leverages its built-in capabilities to process raw text effectively, eliminating the need for extensive preprocessing. BERT's architecture comprehends word context and nuances directly from the text, thanks to its specialized tokenizer. This tokenizer adeptly manages text by splitting it into tokens and inserting necessary special tokens like [CLS] and [SEP], crucial for understanding sequence relationships. Furthermore, BERT inherently accounts for word order through positional embeddings and offers **both case-sensitive and case-insensitive variants**, simplifying our approach to data preparation. As a result, **we forego traditional preprocessing steps**, allowing BERT to fully exploit the textual information in its original form for our NLP tasks. This streamlined preprocessing aligns with BERT's design, facilitating an efficient and nuanced modeling process.

3.6.1.2 Text Representation

We initialize a BERT tokenizer and establish a function for text encoding. Opting for the "uncased" pretrained model variant, which is case-insensitive, further eliminates the necessity for explicit text preprocessing steps. Codes for the steps are as follows:

- **Loading the BERT tokenizer:**

```
# Load the BERT Tokenizer
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
```

```
tokenizer_config.json: 100% 48.0/48.0 [00:00<00:00, 4.22kB/s]
vocab.txt: 100% 232k/232k [00:00<00:00, 1.03MB/s]
tokenizer.json: 100% 466k/466k [00:00<00:00, 35.1MB/s]
config.json: 100% 570/570 [00:00<00:00, 48.7kB/s]
```

- **Defining the encoding function:**

```
# Define the BERT encoding function
def encode_texts(tokenizer, texts, max_length):
    input_ids = []
    attention_masks = []

    for text in texts:
        encoded = tokenizer.encode_plus(
            text,
            add_special_tokens=True,  # Add '[CLS]' and '[SEP]'
            max_length=max_length,    # Pad & truncate all sentences.
            padding='max_length',
            truncation=True,
            return_attention_mask=True,  # Construct attention masks.
            return_tensors='np',  # Return numpy arrays.
        )
        input_ids.append(encoded['input_ids'])
        attention_masks.append(encoded['attention_mask'])

    # Convert lists to arrays
    input_ids = np.concatenate(input_ids, axis=0)
    attention_masks = np.concatenate(attention_masks, axis=0)

    return input_ids, attention_masks
```

- **Encoding the text. Here we use two max sequences length (will be specified later):**

```
# Encodes the text
max_length = 512 #256
train_input_ids, train_attention_masks = encode_texts(tokenizer, X_train.tolist(), max_length)
val_input_ids, val_attention_masks = encode_texts(tokenizer, X_val.tolist(), max_length)
test_input_ids, test_attention_masks = encode_texts(tokenizer, X_test.tolist(), max_length)
```

3.6.1.3 Train-Validation-Test Split

We executed a 60%-20%-20% train-validation-test split, enabling close monitoring of our model's validation performance during training to mitigate overfitting.

3.6.2 BRET-LSTM Neural Network

3.6.2.1 Architecture One - Single LSTM Layer

As our foundational model, we begin with a streamlined architecture that incorporates a single Bidirectional LSTM Layer positioned between the BERT embedding layer and the output layer. This setup utilizes a maximum sequence length of 256. Key components of this architecture are detailed below:

- **Demonstration of Key Components:**

```
# Model configuration
max_length = 256  # Adjust based on your analysis of the text lengths

# Define input layers
input_ids = tf.keras.layers.Input(shape=(max_length,), dtype=tf.int32, name='input_ids')
attention_mask = tf.keras.layers.Input(shape=(max_length,), dtype=tf.int32, name='attention_mask')

# Instantiate and call the custom BERT embedding layer
bert_embedding_layer = BertEmbeddingLayer(bert_model)
sequence_output = bert_embedding_layer((input_ids, attention_mask))

# Continue with the rest of the model
lstm_layer = tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(64, return_sequences=False))(sequence_output)
dropout_layer = tf.keras.layers.Dropout(0.2)(lstm_layer)
classification_layer = tf.keras.layers.Dense(3, activation='softmax')(dropout_layer)
```

3.6.2.2 Architecture Two - Single LSTM Layer with Max Sequence Length set to 512:

To fully leverage BERT's capacity for handling extensive sequences, we adjusted the **maximum sequence length to 512** (the maximum BERT is designed to handle), **reconfigured the BERT tokenizer to match**, and **maintained the architecture described above, setting the max length parameter to 512** in the model configuration. It's important to note, however, that due to resource constraints, we were unable to support a max sequence length of 512 for more complex architectures. Therefore, **for the subsequent architectures featuring additional complexity, we reverted the max sequence length to 256** to ensure compatibility with our GPU resources and continued task execution.

3.6.2.3 Architecture Three - Moderate Multi-Layer Structure:

Building on the foundational architecture outlined in Architecture One, we explored whether integrating more sophisticated structures could enhance our model's performance. To this end, we introduced **additional complexity by incorporating extra dense layers following the Bidirectional LSTM:**

- **Demonstration of Key Components:**

```
# Bi-directional LSTM layer
lstm_layer = tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(64, return_sequences=False))(sequence_output)

# Dropout for regularization
dropout_layer = tf.keras.layers.Dropout(0.2)(lstm_layer)

# MLP for additional representation learning
mlp_layer1 = tf.keras.layers.Dense(128, activation='relu')(dropout_layer)
mlp_dropout1 = tf.keras.layers.Dropout(0.2)(mlp_layer1)
mlp_layer2 = tf.keras.layers.Dense(64, activation='relu')(mlp_dropout1)
mlp_dropout2 = tf.keras.layers.Dropout(0.2)(mlp_layer2)

# Final classification layer
classification_layer = tf.keras.layers.Dense(3, activation='softmax')(mlp_dropout2)
```

3.6.2.4 Architecture Four - Double LSTM-Layer Structure

To investigate the effects of further complexity, we experimented with **adding a second Bidirectional LSTM layer** to our model. Maintaining the core structure defined in Architecture One, we introduced an additional LSTM layer to assess potential performance improvements:

- **Demonstration of Key Components:**

```
# First LSTM layer with return_sequences=True to pass sequences to the next LSTM layer
lstm_layer1 = tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(64, return_sequences=True))(sequence_output)

# Second LSTM layer, processing the output from the first LSTM layer
lstm_layer2 = tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(32, return_sequences=False))(lstm_layer1)

# Dropout for regularization
dropout_layer = tf.keras.layers.Dropout(0.2)(lstm_layer2)

# Classification layer
classification_layer = tf.keras.layers.Dense(3, activation='softmax')(dropout_layer)
```

3.6.3 Model Compiling and Fitting Strategy

Across all four architectures detailed previously, we employed a consistent approach for model compiling and fitting. It's important to note that due to resource constraints, our hyperparameter tuning was confined to layer architecture adjustments, without exploring variations in learning rate or batch size. To closely monitor validation performance and mitigate the risk of overfitting our training set, we adopted a strategy of fitting the model one epoch at a time. The precise number of epochs each architecture underwent will be outlined in the subsequent section, which discusses the test performance of our various architectures.

- **Model Compiling Setup (same for all architectures):**

```
# Compile the model
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=2e-5), loss='sparse_categorical_crossentropy', metrics=['accuracy'])
```

- **Model Fitting Setup (same for all architectures):**

```
history = model.fit(
    [train_input_ids, train_attention_masks],
    y_train,
    validation_data=([val_input_ids, val_attention_masks], y_val),
    epochs=1,
    batch_size=32
)
```

Test Performance Comparison Across Architectures

	Single LSTM Layer Max_Length = 256 # Epoch = 3	**Single LSTM Layer** Max_Length = 512 # Epoch = 3	**Moderate Multi-Layer** Max_Length = 256 # Epoch = 3	**Double LSTM Layer** Max_Length = 256 # Epoch = 2
ACC.	0.7400	0.7463	0.7353	0.7328
Recall	0.7400	0.7463	0.7353	0.7328
F-1 Score	0.7393	0.7484	0.7346	0.7362

Analyzing the test performance across the four architectures, it is evident that **a longer sequence length (512 vs. 256) marginally enhances model performance**. When maintaining a fixed max_length of 256**, incorporating additional complexity—such as multi-layer configurations or double LSTM layers—does not automatically result in improved test metrics.** However, it is worth considering that utilizing a max_length of 512 with these more complex structures could potentially further enhance performance. Due to resource constraints, we were unable to explore this avenue, but it remains a promising area for future investigation when more resources become available.

3.7 Conclusion

Overall, it is clear that our **BERT-LSTM model significantly outperforms the less sophisticated models** discussed in this report for **the task of text classification**. This finding underscores the **BERT-LSTM model's robust ability to adeptly navigate complex NLP challenges.** The enhanced performance can be attributed to BERT's deep understanding of language context and LSTM's proficiency in capturing long-term dependencies. Together, they create a powerful synergy that excels in text analytics tasks, effectively surpassing simpler models by leveraging advanced contextual embeddings and sequential data processing. This superiority reaffirms the value of combining BERT's contextual insights with LSTM's sequential analysis for handling nuanced text classification tasks.

Part 4. Potential Extra Bonus Points:

Our team utilized cross-validation to tune the L2 regularization term for the Logistic Regression model and employed GridSearchCV to fine-tune hyperparameters for the Random Forest model.

Additionally, we deployed the BRET-LSTM model, which significantly outperformed other less sophisticated text analysis methods.

Part 5. Team Meeting Agenda - Team 8

Prepared by: Zijian Zhang
Date and time: 03/02/2024 at 8 pm
Location: Zoom
Team members in attendance: Joyce Xu, Junchen Xia, Yuchen Dai, Zijian Zhang
Meeting objectives: Delegate M2 Tasks
Agenda: Discussing Feedback from M1, Discussing M2 Tasks, Addressing Potential Challenges, and Delegating Tasks for M2
Next Actions:

Action Item	Assigned To	Estimated Hours to Complete This Item	Due Date
M2 Part 1	Joyce Xu	48 hr	03/04/2024
M2 Part 2	Yuchen Dai	48 hr	03/06/2024
M2 Part 3	Junchen Xia, Zijian Zhang	48 hr	03/06/2024

Time meeting ended: 03/02/2024 at 10 pm
Date and time of next meeting: Intended: After M3 posted