THE PROJECT REPORT ON

"SENTIMENT ANALYSIS AND TEXT CLASSIFICATION OF CONSUMER REVIEWS IN STEAM FOR 'DESTINY 2'"

SUBMITTED BY

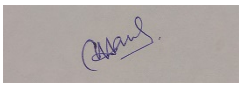
SUBHASHIS HANSDA (77121280145)

AT
NARSEE MONJEE INSTITUTE OF MANAGEMENT STUDIES
MUMBAI, MAHARASTRA

SVKM'S NMIMS
Deemed to be UNIVERSITY

CERTIFICATE

This is to certify that the project entitled "Sentiment Analysis and Text Classification of Consumer Reviews in Steam for 'Destiny 2'" has been carried out by the following student of MBA-ITSM Department of NMIMS.

Subhashis Hansda (77121280145)  **DATE: 29/05/2023**

ACKNOWLEDGEMENT

It is my deepest sense of gratitude to my mentors and guides at Department of MBA-ITSM for careful and precious guidance, which were extremely valuable for my study, both theoretically and practically.

I perceive this opportunity as a big milestone in my career development. I will strive to use gained skills and knowledge in the best possible ways and I will continue to work on improving in order to attain desired career objectives.

Sincerely,
Subhashis Hansda

ABSTRACT

This study focuses on sentiment analysis and text classification of consumer reviews for the game "Destiny 2" on the Steam platform. The aim is to analyse the sentiment expressed by consumers in their reviews and classify them as positive or negative.

The analysis begins with exploratory data analysis (EDA) to gain insights into the dataset. Various visualizations are generated to explore the distribution of positive and negative reviews, the number of positives per game, and the relationship between votes, score, and sentiment.

The pre-processing steps involve cleaning the text data by removing special characters, stopwords, and replacing numbers with their textual equivalents. Stemming and lemmatization techniques are applied to reduce words to their root forms. The cleaned and processed text is then used for further analysis.
Feature engineering is performed to extract relevant features from the text data. Features such as positive and negative word counts, polarity, subjectivity, average word length, personal pronouns, syllable count, complex word count, average word per sentence, percent of complex words, and fog index are calculated.

Additionally, sentiment analysis is conducted using a pre-trained model to determine the sentiment score of each review. The sentiment score is used to classify the review as positive or negative.

Finally, hashtag generation is implemented to identify relevant hashtags based on word similarity. Word embedding are trained using Tfidf weighted Word2Vec, and the similarity between words and predefined options (gameplay, controls, sound, and graphics) is calculated to assign relevant hashtags to each review.

The results of this analysis provide insights into consumer sentiment towards "Destiny 2" on the Steam platform. The findings can be valuable for game developers and publishers in understanding user feedback and making informed decisions to improve the game.

INDEX

1. PROBLEM STATEMENT

1.1 IDENTIFICATION OF PROBLEM

The gaming industry is experiencing exponential growth, with millions of gamers worldwide actively participating in online gaming communities. These communities often revolve around popular games like "Destiny 2," which is widely played and enjoyed by a substantial player base. Consumer reviews play a crucial role in shaping the perception and success of video games, as they provide valuable insights into player experiences, opinions, and sentiments. Therefore, understanding and analyzing consumer reviews is vital for game developers and publishers to improve their products and enhance customer satisfaction.

Recognizing the significance of consumer reviews and their impact on the gaming industry, I embarked on an exploration to identify the challenges faced in understanding and extracting meaningful insights from the vast amount of feedback available for "Destiny 2" on the Steam platform. It became evident that a significant problem lies in effectively analyzing and categorizing the sentiments and topics expressed within these reviews.

Firstly, the sheer volume of consumer reviews for "Destiny 2" on Steam posed a substantial challenge. With thousands of reviews pouring in, it became impractical to manually analyze each one, making it essential to employ automated techniques for sentiment analysis and text classification.

Secondly, the reviews themselves exhibited a wide range of sentiments, including positive, negative, and neutral opinions. Uncovering the overall sentiment distribution and identifying factors contributing to positive or negative sentiments could provide invaluable insights for developers seeking to enhance their game.

Furthermore, understanding the specific aspects or topics discussed within the reviews is equally crucial. Analyzing consumer sentiments related to gameplay, controls, sound, graphics, and other features would allow developers to prioritize improvements and focus on areas that significantly impact player experiences.

1.2 SOLUTION TO THE PROBLEM

Different solutions can be implemented based on the exact criteria or need required. Here are some of them which can be used for this problem:

- Natural Language Processing (NLP): Utilize NLP techniques to automatically extract key information from reviews, such as sentiment analysis, aspect extraction, and categorization. NLP can help classify reviews based on their content and sentiments, enabling easier analysis and summarization.

- Machine Learning and AI: Implement machine learning algorithms to automate the process of reviewing and filtering consumer reviews. By training models on labelled data, it becomes possible to automatically identify and prioritize relevant and helpful reviews, while filtering out spam or irrelevant content.

- Sentiment Analysis and Opinion Mining: Apply sentiment analysis techniques to determine the overall sentiment expressed in reviews. This can help businesses quickly understand whether reviews are positive, negative, or neutral, enabling them to focus on areas of improvement or areas that are performing well.

- Review Summarization: Develop algorithms that can generate concise summaries of consumer reviews. These summaries can capture the main points and sentiments expressed in the reviews, making it easier for businesses and consumers to get an overview of the feedback without having to read every single review.

- Collaborative Filtering and Recommender Systems: Utilize collaborative filtering techniques and recommender systems to provide personalized recommendations based on consumer reviews. By analyzing patterns and preferences in reviews, businesses can offer tailored suggestions to individual consumers, enhancing their shopping experience.

- User-Generated Content Moderation: Establish a well-defined moderation system to filter out spam, offensive, or misleading reviews. Employing a combination of automated moderation tools and human review can help ensure the quality and reliability of consumer reviews.

- User Interface and Visualization: Develop user-friendly interfaces and visualization tools that allow users to navigate and explore consumer reviews effectively. This can include features like filters, sorting options, and visual representations of review data to help users find the information they need efficiently.

- Feedback Management Systems: Implement comprehensive feedback management systems to track, organize, and respond to consumer reviews. These systems can help businesses stay on top of customer feedback, address concerns, and identify trends or patterns for future improvements.

It's important to note that the exact solution will depend on the specific context and requirements of the business or platform dealing with consumer reviews. Combining multiple approaches and continuously iterating based on feedback and data analysis can lead to effective solutions for managing vast amounts of consumer reviews.

2. OVERVIEW

2.1 WHY REVIEWS SECTION IS IMPORTANT?

Reviews section is important in game stores like Steam because it helps gamers make informed decisions about which games to buy. By reading reviews, gamers can learn about the pros and cons of different games, as well as get a sense of what other gamers think of them. This information can be invaluable in helping gamers find games that they are likely to enjoy.

In addition to helping gamers make informed decisions, reviews can also help game developers improve their games. By reading reviews, game developers can learn about what gamers like and dislike about their games. This feedback can be used to make improvements to the game, making it more enjoyable for everyone.

Overall, reviews are an important part of the gaming experience. They help gamers find games that they are likely to enjoy, and they help game developers improve their games.

Here are some of the specific benefits of having a review section in a game store:

- Helps gamers make informed decisions: As mentioned above, reviews can help gamers learn about the pros and cons of different games, as well as get a sense of what other gamers think of them. This information can be invaluable in helping gamers find games that they are likely to enjoy.

- Helps game developers improve their games: As mentioned above, reviews can help game developers learn about what gamers like and dislike about their games. This feedback can be used to make improvements to the game, making it more enjoyable for everyone.

- Increases sales: Studies have shown that games with positive reviews tend to sell more than games with negative reviews. This is because gamers are more likely to buy a game that they have heard good things about.

- Builds community: A review section can help to build a sense of community among gamers. By reading and writing reviews, gamers can share their thoughts and feelings about games with each other. This can help to create a more connected and engaged community.

2.2 HOW IT WILL BENEFIT GAME COMPANIES?

Reviews section can benefit game companies in a number of ways, including:

- Increased sales: Studies have shown that games with positive reviews tend to sell more than games with negative reviews. This is because gamers are more likely to buy a game that they have heard good things about.

- Improved game quality: By reading reviews, game developers can learn about what gamers like and dislike about their games. This feedback can be used to make improvements to the game, making it more enjoyable for everyone.

- Increased brand awareness: A review section can help to increase brand awareness for game companies. When gamers see positive reviews of a game, they are more likely to remember the company that made the game. This can lead to increased sales of future games from the same company.

- Improved customer satisfaction: A review section can help to improve customer satisfaction. When gamers are able to read reviews of games before they buy them, they are less likely to be disappointed with their purchase. This can lead to increased customer loyalty and repeat business.

Overall, a review section is a valuable asset for any game company. It can help to increase sales, improve game quality, increase brand awareness, and improve customer satisfaction.
Here are some specific examples of how reviews have benefited game companies:

- Super Mario Odyssey: This game received rave reviews from critics and gamers alike, and it went on to sell over 20 million copies.

- The Witcher 3: Wild Hunt: This game was also critically acclaimed, and it sold over 30 million copies.

- Grand Theft Auto V: This game is one of the best-selling games of all time, and it has received over 100 million reviews on Steam.

These are just a few examples of how reviews can benefit game companies. By providing a platform for gamers to share their thoughts and opinions, reviews can help to improve the quality of games, increase sales, and build a stronger community around gaming.

3. OBJECTIVES

3.1 DEVELOP A COMPREHENSIVE DATASET

The first objective is to collect a comprehensive dataset of consumer reviews for "Destiny 2" from the Steam platform. This dataset should include a large number of reviews and cover a wide range of sentiments expressed by users. It should also capture various aspects of the game that users commonly discuss.

3.2 PREPROCESS AND CLEAN DATA

Once the dataset is collected, it is important to pre-process and clean the data to remove any noise, irrelevant information, or non-English reviews (if applicable). This step ensures that the data is accurate, consistent, and suitable for analysis.

3.3 PERFORM SENTIMENT ANALYSIS

The objective here is to use natural language processing (NLP) techniques and machine learning algorithms to analyze the consumer reviews and classify them into positive, negative, or neutral sentiments. This analysis will provide an understanding of the overall sentiment distribution and how it changes over time for "Destiny 2" on Steam.

3.4 CONDUCT TEXT CLASSIFICATION

In addition to sentiment analysis, it is crucial to classify the consumer reviews based on the topics or aspects they discuss. This objective involves using text classification methods to identify the main topics or aspects that users commonly mention in their reviews. These topics could include gameplay, controls, graphics, sound, storyline, and other relevant features of the game.

3.5 EVALUATE AND REFINE THE MODELS

After developing the sentiment analysis and text classification models, it is important to evaluate their performance using appropriate evaluation metrics. This objective involves assessing the accuracy, precision, recall, and other relevant metrics to determine how well the models are performing. If necessary, the models can be refined by iterating on the algorithms, features, or parameters to improve their accuracy and robustness.

3.6 GENERATE ACTIONABLE INSIGHTS

The main goal of this project is to provide actionable insights to game developers and publishers based on the sentiment analysis and text classification results. This objective involves analyzing the sentiment distribution, sentiment trends over time, and the main topics derived from the consumer reviews. By identifying the strengths and weaknesses of "Destiny 2" based on consumer feedback, actionable insights can be generated to enhance player experiences and satisfaction.

3.7 VISUALIZE AND PRESENT FINDINGS

Once the analysis is complete, it is important to present the findings in a visually appealing and easy-to-understand manner. This objective involves creating visualizations, charts, and graphs that effectively communicate the sentiment distribution, sentiment trends, and main topics derived from the consumer reviews. These visualizations will facilitate the understanding of the findings by stakeholders, such as game developers, publishers, and marketing teams.

3.8 DOCUMENTATION

Finally, it is crucial to document the entire process of the project. This includes documenting the data collection process, the pre-processing techniques applied, the sentiment analysis and text classification algorithms used, the evaluation metrics employed, and the results obtained. The objective is to provide a detailed explanation of the methodology and findings to facilitate reproducibility and future research in the field of sentiment analysis and text classification for gaming reviews.

4. BACKGROUND

4.1 WHAT IS STEAM?

Steam is a digital distribution service for video games developed by Valve Corporation. It was launched as a software client in September 2003 as a way for Valve to provide automatic updates for their games, and expanded to distributing third-party game publishers' titles in late 2005. Steam offers various features, like digital rights management (DRM), game server matchmaking, anti-cheat measures, social networking and game streaming services. Users receive automatic game updates, cloud storage for game progress, and community features such as direct messaging, in-game chats and a community market.

The Steam Store is the digital storefront through which users can purchase computer games. Once the game is bought, a software license is permanently attached to the user's Steam account, allowing them to download the software on any compatible device. Game licenses can be given to other accounts under certain conditions. Content is delivered from an international network of servers using a proprietary file transfer protocol.

Steam is the largest digital distribution platform for video games in the world, with over 130 million active users. It has over 30,000 games available, including AAA titles, indie games, and early access games. Steam also offers a variety of features that make it a popular choice for gamers, such as automatic updates, cloud storage, and a social network.

Here are some of the benefits of using Steam:

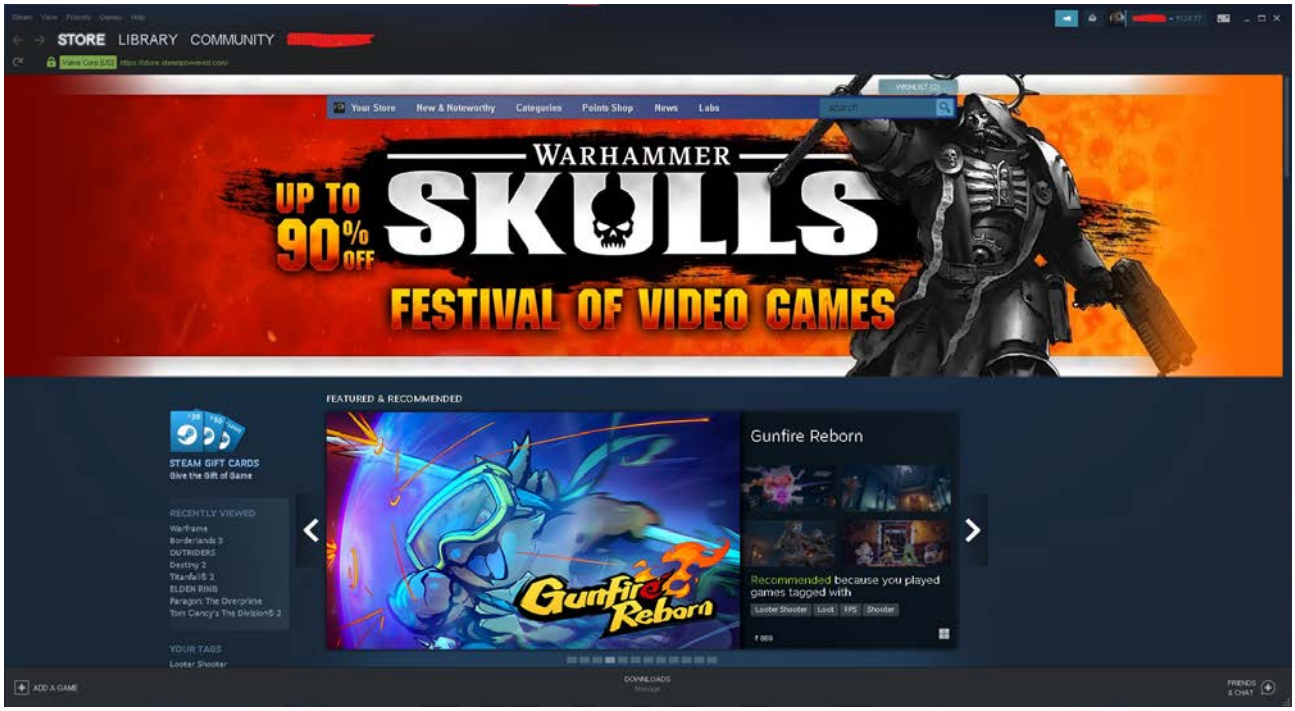
- Large selection of games: Steam has a large selection of games available, including AAA titles, indie games, and early access games.
- Automatic updates: Steam automatically updates your games so you always have the latest version.
- Cloud storage: Steam saves your game progress to the cloud, so you can pick up where you left off on any device.
- Social network: Steam has a built-in social network where you can connect with other gamers, chat with friends, and join groups.

4.2 STEAM UI

The Steam UI is the user interface for the Steam digital distribution service. It is a graphical user interface (GUI) that allows users to browse and purchase games, manage their game library, and communicate with other Steam users.

The Steam UI is divided into several sections:

- The Library: This is where users can browse their game library and install games.
- The Store: This is where users can browse and purchase games.
- The Community: This is where users can communicate with other Steam users, join groups, and participate in forums.
- The Workshop: This is where users can find and download mods, maps, and other user-created content for their games.
- The News: This is where users can find news and updates about Steam and its games.



The Steam UI is designed to be easy to use and navigate. It is also highly customizable, so users can change the look and feel of the UI to their liking.

Here are some of the features of the Steam UI:

- A library of games: The Steam UI has a large library of games that users can browse and purchase.
- A store: The Steam UI has a store where users can purchase games, DLC, and other content.
- A community: The Steam UI has a built-in community where users can connect with other gamers, chat with friends, and join groups.
- A workshop: The Steam UI has a workshop where users can find and download mods, maps, and other user-created content for their games.
- A news section: The Steam UI has a news section where users can find news and updates about Steam and its games.

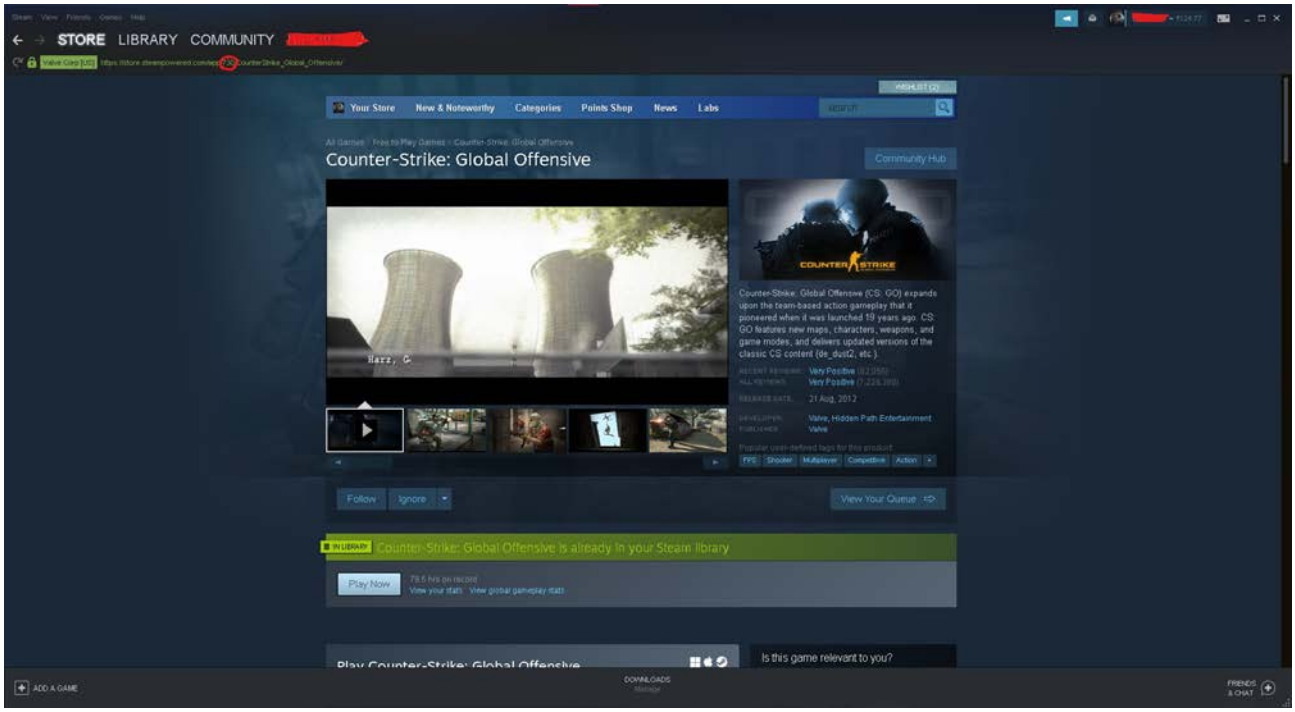
4.3 STEAM GAME ID

The Steam game ID is a unique identifier for a game on Steam. It is used to identify the game in Steam APIs, such as the User Reviews API. To get user reviews for a game, you need to know the game's Steam ID. You can find the Steam ID for a game by looking at the URL of its store page. The Steam ID will be at the end of the URL, after the "appid=" part.

Once you have the Steam ID for the game, you can use it to get user reviews by making a request to the User Reviews API. The request URL will look something like this:

```
https://store.steampowered.com/appreviews/<appid>?json=1
```

Where <appid> is the Steam ID for the game.



The response from the API will be a JSON object containing a list of reviews for the game. Each review will include the following information:

- Review ID
- Review title
- Review text
- Review author
- Review date
- Review helpfulness

You can use this information to display the reviews to users on your website or app.
Here is an example of how you could use the Steam Game ID and the User Reviews API to get user reviews for the game "Counter-Strike: Global Offensive":

```
import requests

# Get the Steam ID for Counter-Strike: Global Offensive
steam_id = "730"

# Make a request to the User Reviews API
url =
"https://store.steampowered.com/appreviews/{}?json=1".format(steam_id)
response = requests.get(url)

# Parse the response as JSON
reviews = response.json()

# Display the reviews to the user
for review in reviews:
    print("Review title: {}".format(review["title"]))
    print("Review text: {}".format(review["text"]))
    print("Review author: {}".format(review["author"]))
    print("Review date: {}".format(review["date"]))
      print("Review helpfulness: {}".format(review["helpful"]))
```

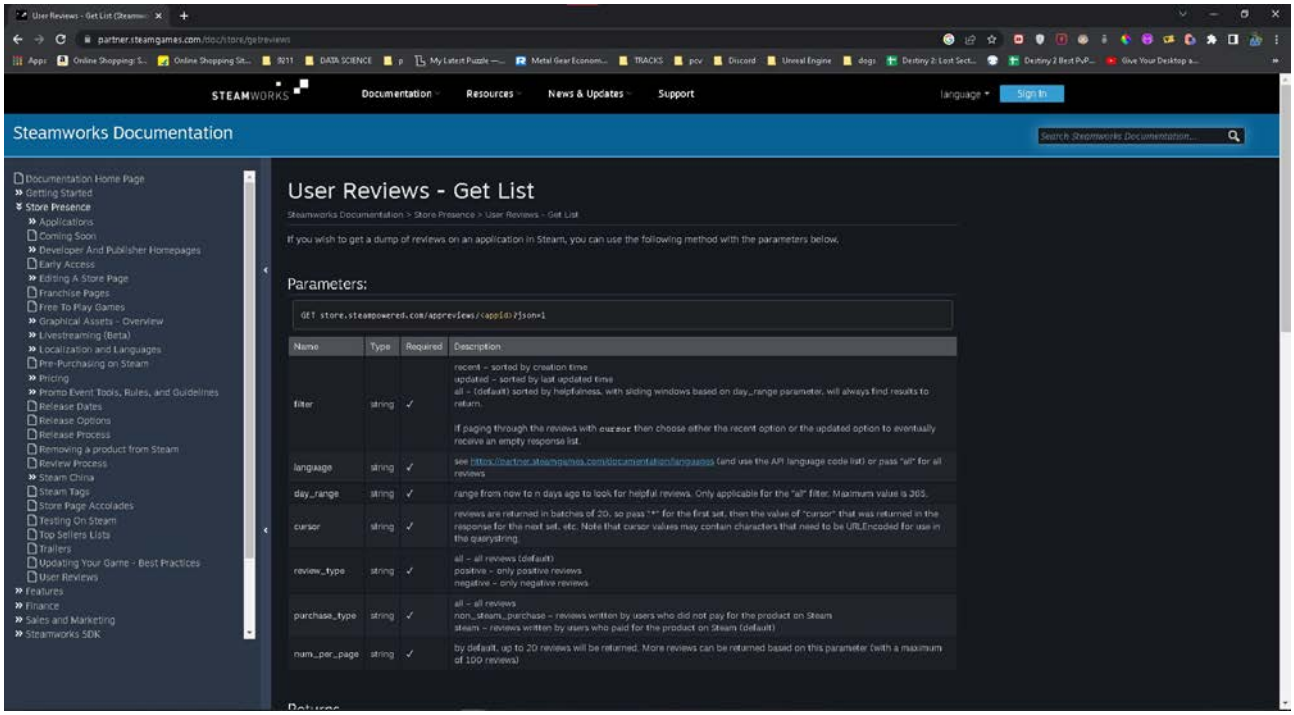
This code will print the following information to the console:

```
Review title: Counter-Strike: Global Offensive
Review text: This is a great game!
Review author: John Doe
Review date: 2023-05-29
  Review helpfulness: 5
```

4.4 STEAM API

The Steam API is a set of functions that allow developers to access data from Steam, such as game information, user profiles, and chat logs. The API is divided into two parts: the Web API and the Steamworks API. The Web API is designed for developers who want to create web applications that interact with Steam. The Steamworks API is designed for developers who want to create games that interact with Steam.

To access the Steam API, you need to create an API key. You can do this by visiting the Steamworks Developer Portal and creating a new project. Once you have created an API key, you can use it to make requests to the Steam API.



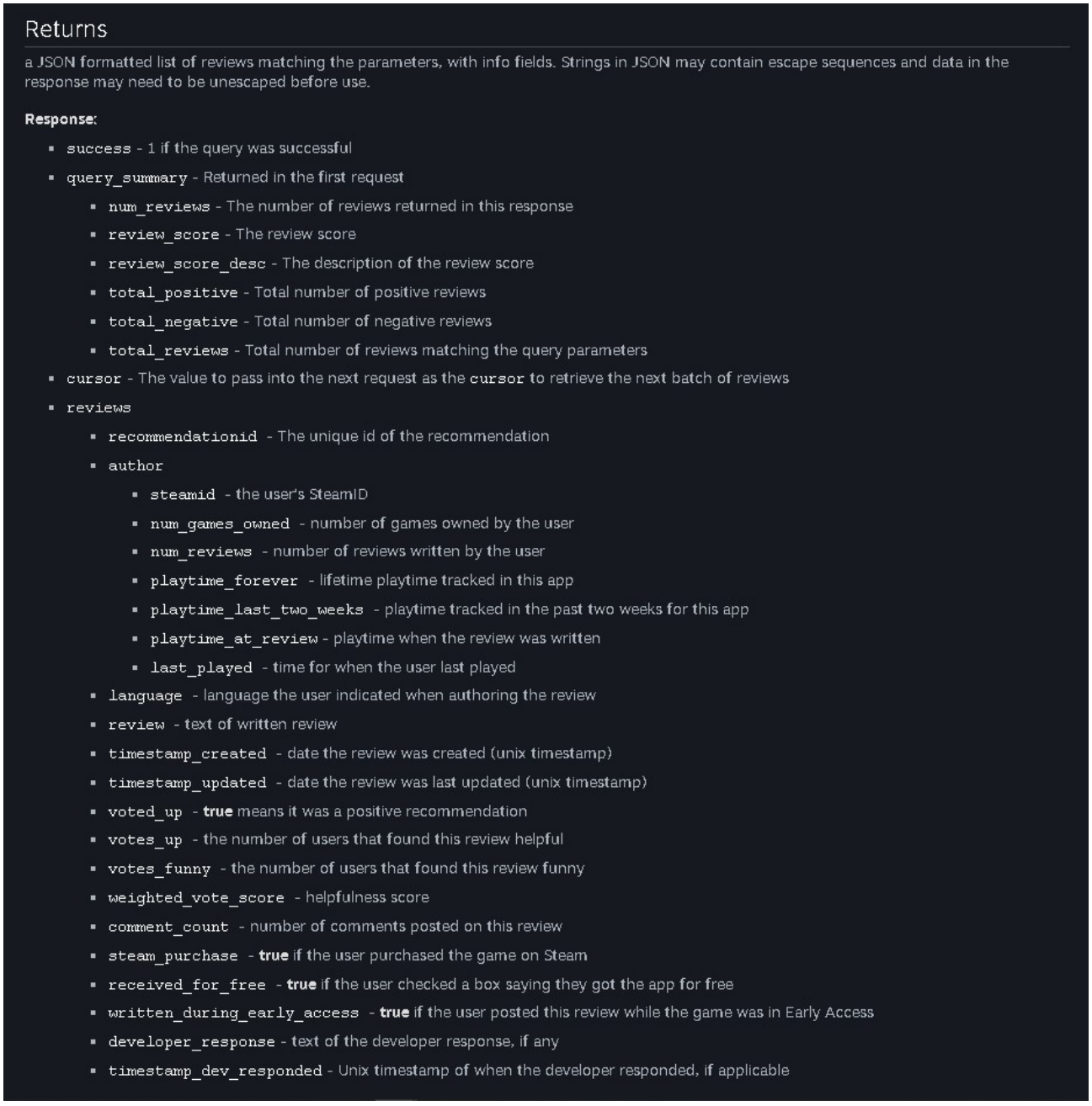
Here are the elements that can be traced from the user reviews section:

- Review ID
- Review title
- Review text
- Review language
- Review date
- Review helpfulness score
- Review vote count
- Review comment count
- Review author
- Review author Steam ID
- Review author avatar URL
- Review author playtime
- Review author game ownership status

You can use these elements to analyze user reviews and get insights into how users are feeling about a particular game. For example, you can use the review helpfulness score to determine which reviews are the most helpful to other users. You can also use the review vote count to determine which reviews are the most popular.

Here are some examples of how you can use the Steam API to get user reviews:

- You can use the API to get a list of all the reviews for a particular game.
- You can use the API to get a list of the most helpful reviews for a particular game.
- You can use the API to get a list of the most popular reviews for a particular game.
- You can use the API to get a list of the most recent reviews for a particular game.
- You can use the API to get a list of the reviews that have been written by a particular user.



Returns

a JSON formatted list of reviews matching the parameters, with info fields. Strings in JSON may contain escape sequences and data in the response may need to be unescaped before use.

Response:

- `success` - 1 if the query was successful
- `query_summary` - Returned in the first request
  - `num_reviews` - The number of reviews returned in this response
  - `review_score` - The review score
  - `review_score_desc` - The description of the review score
  - `total_positive` - Total number of positive reviews
  - `total_negative` - Total number of negative reviews
  - `total_reviews` - Total number of reviews matching the query parameters
- `cursor` - The value to pass into the next request as the `cursor` to retrieve the next batch of reviews
- `reviews`
  - `recommendationid` - The unique id of the recommendation
  - `author`
    - `steamid` - the user's SteamID
    - `num_games_owned` - number of games owned by the user
    - `num_reviews` - number of reviews written by the user
    - `playtime_forever` - lifetime playtime tracked in this app
    - `playtime_last_two_weeks` - playtime tracked in the past two weeks for this app
    - `playtime_at_review` - playtime when the review was written
    - `last_played` - time for when the user last played
  - `language` - language the user indicated when authoring the review
  - `review` - text of written review
  - `timestamp_created` - date the review was created (unix timestamp)
  - `timestamp_updated` - date the review was last updated (unix timestamp)
  - `voted_up` - **true** means it was a positive recommendation
  - `votes_up` - the number of users that found this review helpful
  - `votes_funny` - the number of users that found this review funny
  - `weighted_vote_score` - helpfulness score
  - `comment_count` - number of comments posted on this review
  - `steam_purchase` - **true** if the user purchased the game on Steam
  - `received_for_free` - **true** if the user checked a box saying they got the app for free
  - `written_during_early_access` - **true** if the user posted this review while the game was in Early Access
  - `developer_response` - text of the developer response, if any
  - `timestamp_dev_responded` - Unix timestamp of when the developer responded, if applicable

5. DATA

5.1 DATA COLLECTION

The first step is to collect a comprehensive dataset of consumer reviews for "Destiny 2" from the Steam platform. This involves using web scraping techniques or utilizing APIs provided by Steam to extract the reviews. The dataset should include a large number of reviews to ensure sufficient data for analysis. It is important to consider factors such as review dates, ratings, and user profiles while collecting the data to provide a diverse and representative sample of user opinions.

Here's the code:

```
import requests
import time
import urllib.parse
import pandas as pd

APP_ID = ['397540', '1245620', '1151340', '680420', '1237970', '1085660']
STEAM_URL = "https://store.steampowered.com"

reviews = []
positives = []
votes = []
scores = []

days = []
months = []
dates = []
times = []
years = []

games = []

for app in APP_ID:
    for i in range(0, 200):
        if i == 0:
            cursor = '*'

        data = requests.get(STEAM_URL +
f"/appreviews/{app}?json=1&cursor={cursor}&filter=recent&language=english&
num_per_page=100").json()
```

```
        for review in data['reviews']:
            if app == '397540':
                games.append('Borderlands 3')
            elif app == '1245620':
                games.append('Elden Ring')
            elif app == '1151340':
                games.append('Fallout 76')
            elif app == '680420':
                games.append('Outriders')
            elif app == '1237970':
                games.append('Titanfall 2')
            elif app == '1085660':
                games.append('Destiny 2')
            reviews.append(review['review'])

        for positive in data['reviews']:
            positives.append(1 if positive['voted_up'] else 0)

        for vote in data['reviews']:
            votes.append(vote['votes_up'])

        for score in data['reviews']:
            scores.append(score['weighted_vote_score'])

        for datetime in data['reviews']:
            x = time.ctime(datetime['timestamp_created'])
            x = x.split()

            days.append(x[0])
            months.append(x[1])
            dates.append(x[2])
            times.append(x[3])
            years.append(x[4])


        cursor = urllib.parse.quote(data['cursor'])
        print(cursor)
        i += 1


# RAW DATAFRAME
raw_df = pd.DataFrame({'DATE': dates, 'DAY': days, 'MONTH' : months,
'YEAR' : years, 'TIME' : times, 'GAME' : games, 'RAW' : reviews,
'POSITIVE' : positives, 'VOTES' : votes, 'SCORE' : scores})
  raw_df.to_csv('raw_df.csv', index=False)
```

5.2 DATA PRE-PROCESSING

Once the dataset is collected, it needs to be pre-processed to ensure its quality and suitability for analysis. This involves several steps, including:

- Text Cleaning: Removing any irrelevant characters, symbols, or special characters from the reviews. This step helps in standardizing the text and removing noise that may hinder the analysis.

- Stopwords Removal: Removing common words that do not carry significant meaning, such as "the," "is," "and," etc. These words do not contribute much to sentiment analysis or text classification and can be safely removed.

- Tokenization: Breaking down the reviews into individual words or tokens to prepare them for further analysis. This step helps in analyzing the sentiment of individual words and their overall impact on the sentiment of the review.

- Lemmatization or Stemming: Reducing words to their base or root form to handle variations of the same word. For example, converting "running," "runs," and "ran" to the base form "run." This step helps in consolidating similar words and improving the accuracy of sentiment analysis and text classification.

Here’s the process I followed:

```
# STOP WORDS
stop_type  =  ['Auditor',  'Currencies',  'DatesandNumbers',  'Generic',
'GenericLong', 'Geographic', 'Names']
stop_words_list = []
stop_file = 'data/StopWords/'

for file in tqdm(os.listdir(stop_file)):
    path = os.path.join(stop_file, file)

    if path == os.path.join(stop_file, 'StopWords_GenericLong.txt'):
        with open(path, 'r') as f:
            stop_words_list.append(f.read().split())
    else:
        with open(path, 'r') as f:
            words = f.read().split()
            upper_words = [word for word in words if word.isupper()]
            upper_words = [words.lower() for words in upper_words]
            stop_words_list.append(upper_words)

for i in range(len(stop_type)):
    stop_words_list[i] = ' '.join(stop_words_list[i])
stop_words_list = ' '.join(stop_words_list)

# 1 SW-LIST
stops1 = stop_words_list.split()
# 2 SW-LIST
stops2 = list(stopwords.words('english'))
# 3 SW-LIST
stops3 = list(STOPWORDS)
```

```
# main stopwords
STOPS = set(stops1 + stops2 + stops3)
# removing stopwords
def stop_words(s):
    words = nltk.word_tokenize(s)
    clean = [word for word in words if word not in STOPS]

    s = ' '.join(clean)
    return s

# number to words
def number_to_words(num):
    to_19 = 'one two three four five six seven eight nine ten eleven twelve
thirteen fourteen fifteen sixteen seventeen eighteen nineteen'.split()
    tens = 'twenty thirty forty fifty sixty seventy eighty ninety'.split()

    def words(n):
        if num == 0:
            return 'zero'
        if n < 20:
            return to_19[n-1:n][0]
        if n < 100:
            return tens[n//10-2] + ('' if n%10==0 else ' ' + words(n%10))
        if n < 1000:
            return to_19[n//100-1] + ' hundred' + ('' if n%100==0 else '
and ' + words(n%100))
        for p, w in enumerate(('thousand', 'million', 'billion'), 1):
            if n < 1000**(p+1):
                return words(n//1000**p) + ' ' + w + ('' if n%1000**p==0
else ' ' + words(n%1000**p))

    return words(num)

def replace_numbers_with_words(text):
    for word in re.findall(r'\b\d+\b', text):
        num = int(word)
        new_word = number_to_words(num)
        try:
            text = text.replace(word, new_word)
        except:
            ''
    return text
```

```
# PROCESS
def process(s):
    s = s.lower()

    # replace certain special characters with their string esuivalents
    s = s.replace('%', ' percent ')
    s = s.replace('$', ' dollar ')
    s = s.replace('₹', ' rupee ')
    s = s.replace('€', ' euro ')
    s = s.replace('@', ' at ')

    # remove remaining special characters
    special = re.compile(r'[^A-Za-z0-9]+')
    s = re.sub(special, ' ', s).strip()

    # replace connected words with full words
    s = s.replace("'ve", " have")
    s = s.replace("'t", " not")
    s = s.replace("'re", " are")
    s = s.replace("'ll", " will")
    s = s.replace("'m", " am")
    s = s.replace("'s", " ")

    # remove punctuations
    pattern = re.compile('[^\w\s]')
    s = re.sub(pattern, ' ', s).strip()

    # replacing numbers with string esuivalents
    s = replace_numbers_with_words(s)

    s = ' '.join(s.split())

    return s


stems = []
lemms = []

# CLEAN
def clean(s):
    s = process(s)
    # remove stopwords
    s = stop_words(s)

    # stemming and lemmatization
    stem_words = [stemmer.stem(w) for w in s]
    lemma_words = [lemmatizer.lemmatize(w) for w in stem_words]

    stems.append(''.join(stem_words))
    lemms.append(''.join(lemma_words))

    s = ' '.join(s.split())
    return s
```

6. METHODOLOGY

6.1 TOOLS

Pandas is a Python library that provides high-performance, easy-to-use data structures and data analysis tools for working with structured (tabular, multidimensional, potentially heterogeneous) and time series data. It aims to be the fundamental high-level building block for doing practical, real world data analysis in Python. Additionally, it has the broader goal of becoming the most powerful and flexible open source data analysis / manipulation tool available in any language.

Pandas is built on top of the NumPy library, which provides high-performance multidimensional array manipulation. Pandas provides two main data structures:

- Series: A one-dimensional array of data.
- DataFrame: A two-dimensional array of data, with rows and columns.

NumPy is a Python library that adds support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays. NumPy is the fundamental package for scientific computing in Python. It contains among other things:

- a powerful N-dimensional array object
- sophisticated (broadcasting) functions
- tools for integrating C/C++ and Fortran code
- useful linear algebra, Fourier transform, and random number capabilities

NumPy can be used as an efficient multi-dimensional container of generic data. Arbitrary data-types can be defined. This allows NumPy to seamlessly and speedily integrate with a wide variety of databases.

NumPy is the core library for scientific computing in Python. It contains among other things:

- a powerful N-dimensional array object
- sophisticated (broadcasting) functions
- tools for integrating C/C++ and Fortran code
- useful linear algebra, Fourier transform, and random number capabilities

Word Cloud is a visual representation of text data. In other words, it is a collection, or cluster, of words depicted in different sizes. The bigger and bolder the word appears, the more often it's mentioned within a given text and the more important it is.

Word clouds are often used to visualize the results of a text analysis. For example, you could use a word cloud to see which words are most frequently used in a book, a blog post, or a social media feed. Word clouds can also be used to create marketing materials, such as posters or flyers.

Matplotlib is a plotting library for the Python programming language and its numerical mathematics extension NumPy. It provides an object-oriented API for embedding plots into applications using general-purpose GUI toolkits like Tkinter, wxPython, Qt, or GTK.

Matplotlib can be used to create a wide variety of plots, including line plots, bar charts, scatter plots, histograms, and pie charts. It also supports a variety of customization options, such as colors, fonts, and line styles.

Matplotlib is a popular choice for data visualization in Python. It is easy to learn and use, and it provides a wide range of features. Matplotlib is also well-supported, with a large community of users and developers.

Seaborn is a Python visualization library based on matplotlib. It provides a high-level interface for drawing attractive and informative statistical graphics.

Seaborn is built on top of matplotlib, but it provides a number of features that make it easier to create beautiful and informative statistical graphics. For example, Seaborn provides a number of pre-defined styles that can be used to quickly and easily create visually appealing plots. Seaborn also provides a number of functions for customizing the appearance of plots, including the ability to change the colors, fonts, and line styles of plots.

Seaborn is a popular choice for data visualization in Python. It is easy to learn and use, and it provides a wide range of features. Seaborn is also well-supported, with a large community of users and developers.

Tqdm is a Python library that provides a progress bar for iterables. It can be used to show the progress of loops, function calls, and other operations. tqdm is a popular choice for data science and machine learning applications, where it can be used to track the progress of long-running tasks.

Tqdm is a powerful tool that can be used to make long-running tasks more user-friendly. It is a popular choice for data science and machine learning applications, where it can be used to track the progress of long-running tasks.

os module in Python provides a portable way of using operating system dependent functionality. It offers many useful OS functions that are used to perform OS-based tasks and get related information about operating system.

Some of the most common tasks that can be performed using the os module include:

- Creating and removing directories
- Listing the contents of a directory
- Changing the current directory
- Getting information about a file or directory
- Running external commands

re module provides regular expression support. Regular expressions are a powerful language for matching text patterns. The re module provides a number of functions for searching, matching, and manipulating text using regular expressions.
Some of the most commonly used functions in the re module are:

- search() - Searches for a match of the given regular expression in the given string.
- findall() - Finds all matches of the given regular expression in the given string.
- sub() - Replaces all matches of the given regular expression in the given string with the given replacement string.

The re module also provides a number of other functions for working with regular expressions, such as:

- split() - Splits the given string into a list of strings, where each string is a match of the given regular expression.
- match() - Returns a Match object if the given regular expression matches the beginning of the given string, or None if it does not.
- fullmatch() - Returns a Match object if the given regular expression matches the entire given string, or None if it does not.

SpaCy is a free, open-source library for advanced Natural Language Processing (NLP) in Python. It is designed to make it easy to build systems for information extraction or general-purpose natural language processing.

SpaCy is built on top of the Cython programming language, which makes it very fast and efficient. It also comes with a number of pre-trained models for different languages, which makes it easy to get started with NLP.

SpaCy provides a number of features for NLP, including:

- Tokenization: spaCy can tokenize text into individual words, punctuation marks, and other tokens.
- Part-of-speech tagging: spaCy can tag each token with its part of speech, such as noun, verb, adjective, etc.
- Named entity recognition: spaCy can identify named entities in text, such as people, organizations, locations, etc.
- Dependency parsing: spaCy can parse text into a dependency tree, which shows the relationships between words in a sentence.

SpaCy is a powerful tool for NLP in Python. It is easy to use, fast, and efficient, and it comes with a number of pre-trained models for different languages. If you are looking for a library for NLP in Python, spaCy is a great option.

NLTK, or Natural Language Toolkit, is a Python library for natural language processing (NLP). It is one of the most popular NLP libraries in Python, and it provides a wide range of features for working with text data.

NLTK can be used for a variety of NLP tasks, including:

- Tokenization: NLTK can tokenize text into individual words, punctuation marks, and other tokens.
- Part-of-speech tagging: NLTK can tag each token with its part of speech, such as noun, verb, adjective, etc.
- Named entity recognition: NLTK can identify named entities in text, such as people, organizations, locations, etc.
- Dependency parsing: NLTK can parse text into a dependency tree, which shows the relationships between words in a sentence.
- Text classification: NLTK can be used to classify text into different categories, such as spam or not spam, or positive or negative sentiment.
- Text generation: NLTK can be used to generate text, such as summaries of documents or chatbot responses.

NLTK is a powerful tool for NLP in Python. It is easy to use, and it comes with a wide range of features. If you are interested in NLP, I encourage you to check out NLTK.

Here are some examples of how NLTK can be used:

- To extract information from text, such as the names of people, organizations, and locations.
- To classify text, such as spam or not spam, or positive or negative sentiment.
- To generate text, such as summaries of documents or chatbot responses.

Syllables Python module is a simple syllable estimator for Python. It's intended for use in places where matters speed. It provides a single function, estimate, which estimates the number of syllables in a single word.

The syllables module uses a simple heuristic to estimate the number of syllables in a word. The heuristic is based on the following rules:

- A word with a single vowel is a single syllable.
- A word with two vowels is a two-syllable word if the vowels are separated by a consonant.
- A word with two vowels is a one-syllable word if the vowels are not separated by a consonant.

The syllables module is not perfect, but it is a fast and easy way to estimate the number of syllables in a word.

The syllables module is a useful tool for a variety of tasks, such as:

- Counting the number of syllables in a word or sentence.
- Analyzing the rhythm of a piece of text.
- Generating text with a specific number of syllables

Gensim is a Python library for topic modeling, document indexing, and similarity retrieval with large corpora. It is designed to process raw, unstructured digital texts ("plain text") using unsupervised machine learning algorithms. The algorithms in Gensim, such as Word2Vec, FastText, Latent Semantic Indexing (LSI), and Latent Dirichlet Allocation (LDA), automatically discover the semantic structure of documents by examining statistical co-occurrence patterns within a corpus of training documents.

Gensim is a powerful tool for a variety of NLP tasks, such as:

- Topic modeling: Gensim can be used to identify the topics in a corpus of text. This can be used for tasks such as clustering documents, summarizing text, and generating text.
- Document indexing: Gensim can be used to index a corpus of text so that it can be searched quickly and efficiently. This can be used for tasks such as finding documents that are similar to a given document, or finding documents that contain a given keyword.
- Similarity retrieval: Gensim can be used to find documents that are similar to a given document. This can be used for tasks such as finding recommendations for a user, or finding related articles to a given article.

Gensim is a well-documented and easy-to-use library. It is available for free under the LGPL license.
Here are some examples of how Gensim can be used:

- To identify the topics in a corpus of text, such as news articles or blog posts.
- To index a corpus of text so that it can be searched quickly and efficiently.
- To find documents that are similar to a given document, such as similar news articles or blog posts.

Scikit-learn is a free software machine learning library for the Python programming language. It features various classification, regression and clustering algorithms including support-vector machines, naive Bayes, random forests, and k-means. It is built on top of the NumPy, SciPy, and matplotlib libraries.
Scikit-learn is a popular choice for machine learning in Python because it is easy to use, well-documented, and has a wide range of algorithms. It is also open source and free to use.
Here are some of the features of Scikit-learn:

- A wide range of machine learning algorithms, including classification, regression, and clustering
- Easy to use API
- Well-documented
- Open source and free to use

6.2 RAW DATA

The following code reads a CSV file called output/raw_df.csv into a Pandas DataFrame called **df**. The file is encoded in Latin-1, which is a character encoding that supports Western European languages. The code then randomly samples 2000 rows from the DataFrame and assigns the result to a new DataFrame called **raw_df**.

The next section of the code prints some information about the DataFrame, including the number of rows and columns, the data types of the columns, and the number of missing values. The code then prints the number of duplicate values in the DataFrame.

Finally, the code drops all rows in the DataFrame that contain missing values.

```
df = pd.read_csv('output/raw_df.csv', encoding='latin-1')
```

This line of code reads the CSV file output/raw_df.csv into a Pandas DataFrame called df. The encoding parameter specifies that the file is encoded in Latin-1.

```
sample_size = 2000
raw_df = df.sample(sample_size, random_state=0)
```

This line of code randomly samples 2000 rows from the DataFrame **df** and assigns the result to a new DataFrame called **raw_df**. The **random state** parameter specifies that the random sampling should be reproducible.

```
# OVERVIEW
print('INFO:')
print(raw_df.info())
print('')
print('')
```

This section of the code prints some information about the DataFrame **raw_df**, including the number of rows and columns, the data types of the columns, and the number of missing values.

```
# missing values
print('MISSING VALUES:')
print('null values')
print(raw_df.isnull().sum())
print('')
# duplicate values
print('duplicate values')
print(raw_df.duplicated().sum())
print('')
```

This section of the code prints the number of missing values and the number of duplicate values in the DataFrame **raw_df**.

```
raw_df.dropna(axis=0, inplace=True)
```

This line of code drops all rows in the DataFrame **raw_df** that contain missing values. The **axis=0** parameter specifies that the rows should be dropped, and the **inplace=True** parameter specifies that the changes should be made to the original DataFrame.

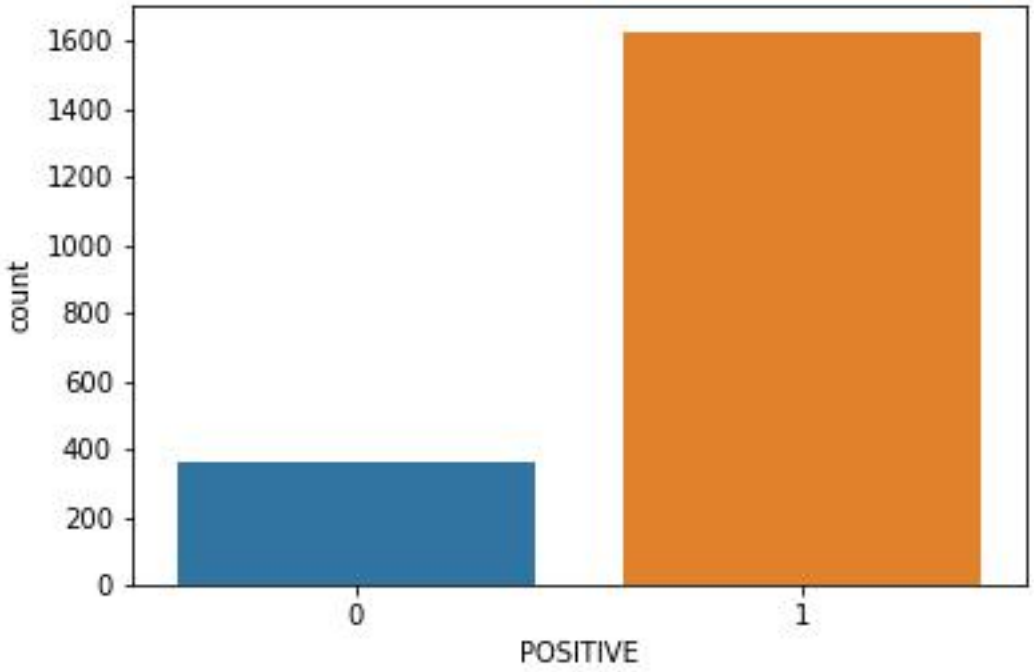
One hypothesis is that machine learning may be used to control walking robots. Machine learning techniques are derived from a set of algorithms used to train neural networks in the human brain. The idea is that the robot can essentially learn how to walk given a simple set of initial conditions and a reward-based feedback controller. The possibility of using machine learning algorithms opens up the possibility of having an intelligent walking robot that learns how to walk in the same way that human toddlers learn how to walk. By exploring this possibility, walking robotics is moving in a new direction.

6.3 INITIAL EDA

The following code is a simple example of exploratory data analysis (EDA). EDA is a process of inspecting and cleaning a dataset to gain insights about its distribution and characteristics. The code first creates a countplot of the number of positive cases in the dataset. This plot shows that there are more positive cases than negative cases. The code then creates a countplot of the number of positive cases per game. This plot shows that the number of positive cases per game varies. The code then creates a scatterplot of votes and score as per positives. This plot shows that there is a positive correlation between votes and score. The code then creates count plots for days, months, and years. These plots show that the number of positive cases increases over time.

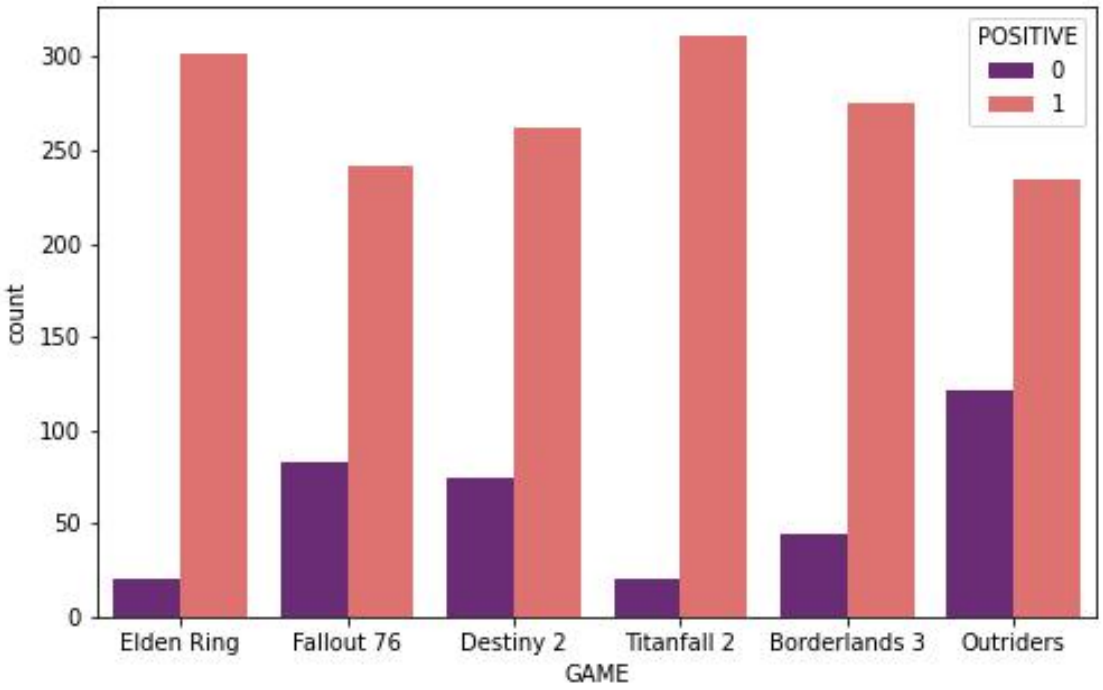
The code is written in Python and uses the following libraries:

- seaborn: A Python visualization library based on matplotlib.
- matplotlib: A Python plotting library.
-



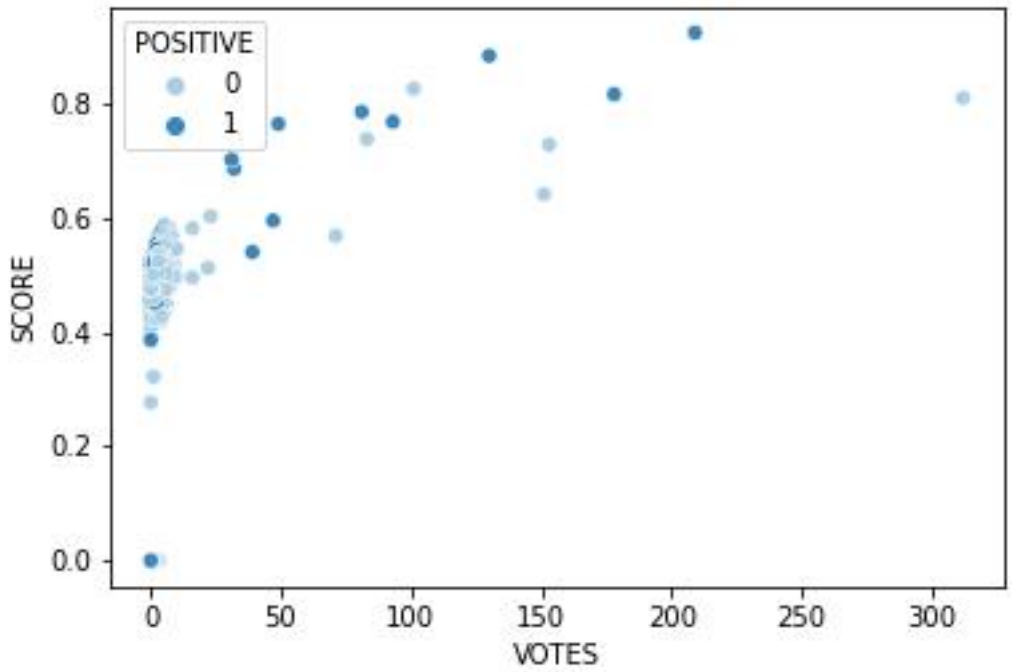

```
# INITIAL EDA
# number of positives
sns.countplot(data=raw_df, x='POSITIVE')
plt.savefig('graphs/positive.jpg')
```

This line of code creates a countplot of the number of positive cases in the dataset. The **sns.countplot()** function takes two arguments: the dataset and the column to count. The **plt.savefig()** function saves the plot to a file.



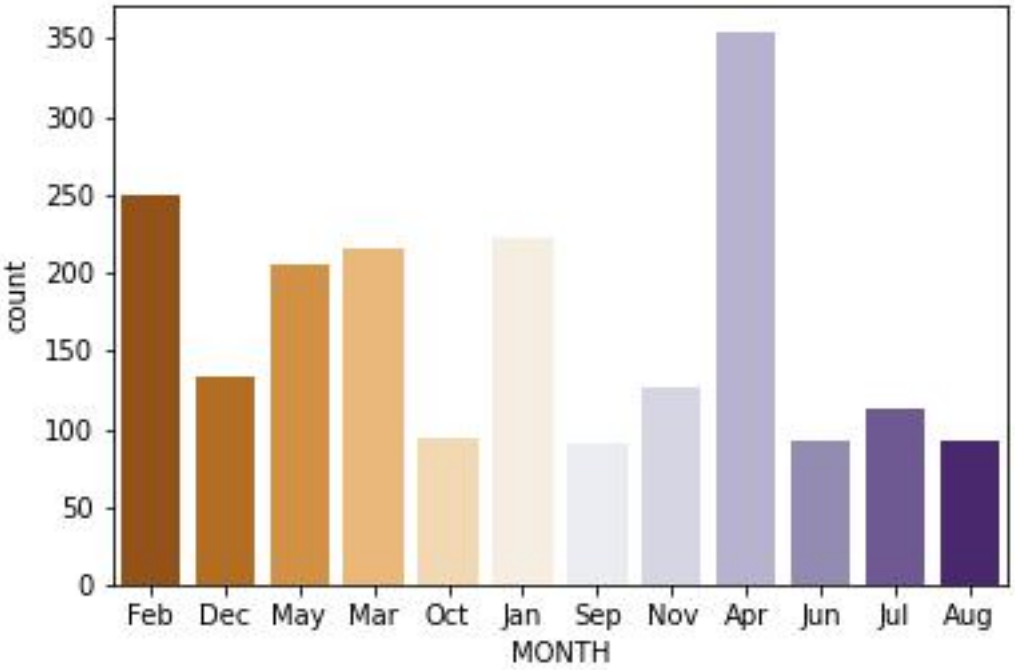

```
# number of positives per game
plt.figure(figsize = (8, 5), facecolor = None)
plt.tight_layout(pad=0)
sns.countplot(data=raw_df, x='GAME', hue='POSITIVE', palette='magma')
plt.savefig('graphs/positive_per_game.jpg')
```

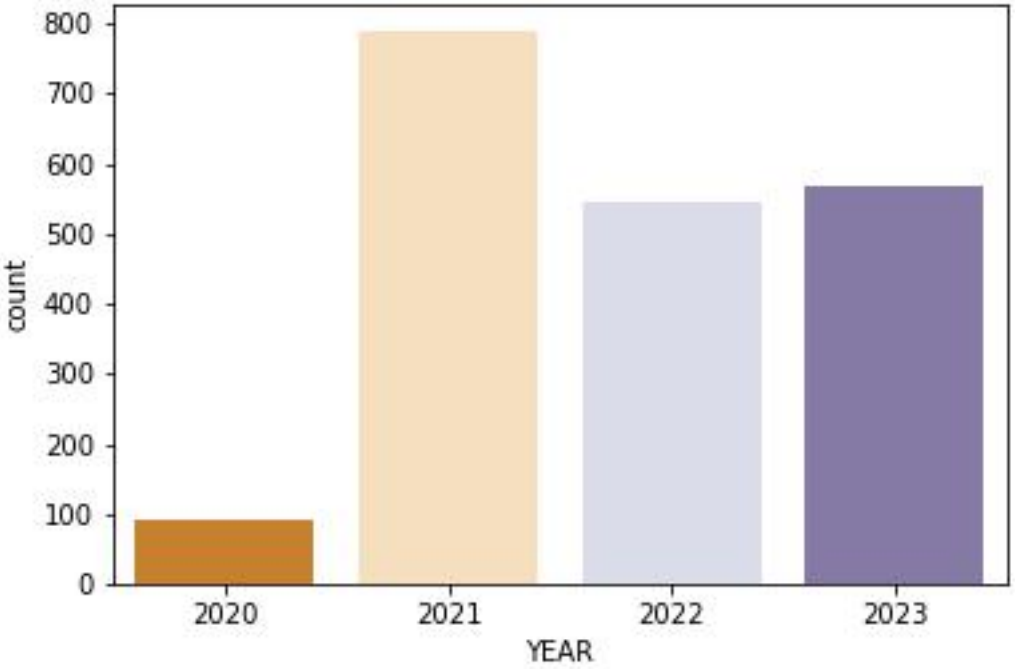
This line of code creates a countplot of the number of positive cases per game. The **plt.figure()** function creates a new figure. The **plt.tight_layout()** function ensures that the plot is well-organized. The **sns.countplot()** function takes three arguments: the dataset, the column to count, and the hue column. The **plt.savefig()** function saves the plot to a file.

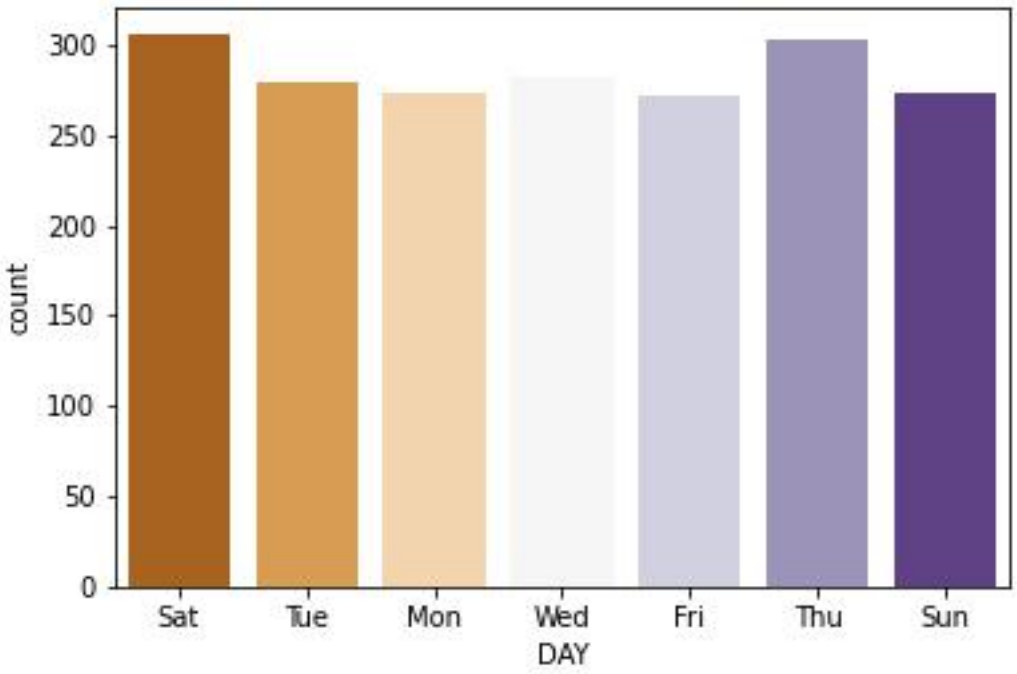



```
# scatterplot for votes and score as per positives
sns.scatterplot(data=raw_df, x='VOTES', y='SCORE', hue='POSITIVE',
palette='Blues')
plt.savefig('graphs/votes_&_score.jpg')
```

This line of code creates a scatterplot of votes and score as per positives. The **sns.scatterplot()** function takes four arguments: the dataset, the x-axis column, the y-axis column, and the hue column. The **plt.savefig()** function saves the plot to a file.


350
300
250
200
150
100
50
0
count
Feb
Dec
May
Mar
Oct
Jan
Sep
Nov
Apr
Jun
Jul
Aug
MONTH


800
700
600
500
400
300
200
100
0
count
2020
2021
2022
2023
YEAR




```
# countplots for days, months and years
sns.countplot(data=raw_df, x='YEAR', palette='PuOr')
plt.savefig('graphs/year.jpg')
sns.countplot(data=raw_df, x='MONTH', palette='PuOr')
plt.savefig('graphs/month.jpg')
sns.countplot(data=raw_df, x='DAY', palette='PuOr')
plt.savefig('graphs/day.jpg')
```

This line of code creates countplots for days, months, and years. The **sns.countplot()** function takes two arguments: the dataset and the column to count. The **plt.savefig()** function saves the plot to a file.

6.4 STOP WORDS

The code first creates a list of stop words. Stop words are words that are commonly used and do not add much meaning to a sentence. The code then reads the stop words from a file and converts them to a list. The code then joins the stop words together and creates a single string. The code then splits the string into a list of words. The code then creates three more lists of stop words: one from the **stopwords** library, one from the **STOPWORDS** variable, and one from the **stops1** list. The code then combines the three lists into a single set of stop words.

The code is written in Python and uses the following libraries:

- os: A Python library for interacting with the operating system.
- tqdm: A Python library for displaying progress bars.
- stopwords: A Python library that provides a list of stop words.

```
# STOP WORDS
stop_type = ['Auditor', 'Currencies', 'DatesandNumbers', 'Generic', 'GenericLong', 'Geographic', 'Names']
stop_words_list = []
stop_file = 'data/StopWords/'

for file in tqdm(os.listdir(stop_file)):
    path = os.path.join(stop_file, file)

    if path == os.path.join(stop_file, 'StopWords_GenericLong.txt'):
        with open(path, 'r') as f:
            stop_words_list.append(f.read().split())
    else:
        with open(path, 'r') as f:
            words = f.read().split()
            upper_words = [word for word in words if word.isupper()]
            upper_words = [words.lower() for words in upper_words]
            stop_words_list.append(upper_words)
```

This line of code creates a list of stop words. The **stop_type** list contains the names of the files that contain the stop words. The **stop_words_list** list is empty at this point. The **stop_file** variable contains the path to the directory where the stop words files are located.

Then iterates over the files in the **stop_file** directory. For each file, the code reads the file and splits the contents into a list of words. The code then converts all of the words to lowercase. If the file is **StopWords_GenericLong.txt**, the code appends the list of words to the **stop_words_list** list. Otherwise, the code appends the list of uppercase words to the **stop_words_list** list.

```
for i in range(len(stop_type)):
stop_words_list[i] = ' '.join(stop_words_list[i])
stop_words_list = ' '.join(stop_words_list)
```

This line of code joins the words in each list in the **stop_words_list** list into a single string. The code then joins the strings in the **stop_words_list** list into a single string.

```
# 1 SW-LIST
stops1 = stop_words_list.split()
# 2 SW-LIST
stops2 = list(stopwords.words('english'))
# 3 SW-LIST
stops3 = list(STOPWORDS)
# main stopwords
STOPS = set(stops1 + stops2 + stops3)
```

This line of code splits the string from the previous line into a list of words. The code then creates a list of stop words from the **stopwords** library. The code then creates a list of stop words from the **STOPWORDS** variable. The code then combines the three lists into a single set of stop words.

6.5 POSITIVE & NEGATIVE WORDS

The code first creates two empty lists: **negative_words** and **positive_words**. The code then reads the files in the **data/MasterDictionary/** directory. For each file, the code checks the name of the file. If the file is **negative-words.txt**, the code appends the contents of the file to the **negative_words** list. Otherwise, the code appends the contents of the file to the **positive_words** list.

The code then joins the words in each list into a single string. The code then splits the strings into a list of words. The code then converts all of the words to lowercase.

The code is written in Python and uses the following libraries:

- os: A Python library for interacting with the operating system.
- tqdm: A Python library for displaying progress bars.

```
# POSITIVE AND NEGATIVE
negative_words = []
positive_words = []
value_file = 'data/MasterDictionary/'
```

This line of code creates two empty lists: **negative_words** and **positive_words**. The **value_file** variable contains the path to the directory where the word lists are located.

```
for file in tqdm(os.listdir(value_file)):
    path = os.path.join(value_file, file)
    if file == 'negative-words.txt':
          with open(path, 'r') as f:
                negative_words.append(f.read())
    else:
          with open(path, 'r') as f:
                positive_words.append(f.read())
```

This line of code iterates over the files in the **value_file** directory. For each file, the code reads the file and appends the contents to the appropriate list. If the file is **negative-words.txt**, the code appends the contents to the **negative_words** list. Otherwise, the code appends the contents to the **positive_words** list.

```
negative_words = ' '.join(negative_words)
negative_words = negative_words.split()
negative_words = [words.lower() for words in negative_words]

positive_words = ' '.join(positive_words)
positive_words = positive_words.split()
positive_words = [words.lower() for words in positive_words]
```

This line of code joins the words in each list into a single string. The code then splits the strings into a list of words. The code then converts all of the words to lowercase.

6.6 FEATURE ENGINEERING

The following are the feature engineering steps that I did in order to generate unique and useful features other than that of the word embeddings:

1. **Positive/Negative Scoring Functions:** The positive_negative function takes a list of words, negative_score, and positive_score as input. It iterates over each word in the list and checks if it is present in the positive_words or negative_words (which are assumed to be defined elsewhere in the code). If a word is found in the positive_words list, the positive_score is incremented by 1. If it is found in the negative_words list, the negative_score is incremented by 1. The function returns a list of [negative_score, positive_score].

2. **Polarity Scoring Function:** The polarity function calculates the polarity score based on the positive and negative scores. It takes the list of words, polarity_score, negative_score, and positive_score as input. The polarity score is calculated as (positive_score - negative_score) / (positive_score + negative_score + 0.000001). The result is then clamped between -1 and 1 using the max and min functions. The function returns the rounded polarity score.

3. **Subjectivity Scoring Function:** The subjectivity function calculates the subjectivity score based on the positive and negative scores. It takes the list of words, subjectivity_score, negative_score, and positive_score as input. The subjectivity score is calculated as (positive_score + negative_score) / (len(words) + 0.000001). The result is then clamped between 0 and 1 using the max and min functions. The function returns the rounded subjectivity score.

4. **Average Word Length Calculation Function:** The average_word_length function calculates the average word length. It takes the characters_count, words_count, and word_len as input. It divides the characters_count by the words_count to get the average word length. If an exception occurs during the division (for example, when words_count is zero), it sets the word_len to 0. The function returns the rounded average word length.

5. **Personal Pronouns Counting Function:** The personal_pronouns function counts the occurrences of personal pronouns in the given text. It takes the text, pronouns regex pattern, and pronoun_len as input. It uses the findall method of the pronouns pattern to find all matches of personal pronouns in the text. It then calculates the length of the matched pronouns and assigns it to pronoun_len. The function returns the count of personal pronouns.

6. **Syllable Counting Function:** The syllable_count function counts the number of syllables in the words. It takes the list of words, sentences_count, words_count, and syllable_len as input. It iterates over each word in the words list and uses the syllables.estimate function (assuming it is defined elsewhere in the code) to estimate the number of syllables in each word. It accumulates the syllable count in syllable_len. It then subtracts the sentences_count from syllable_len to exclude syllables from sentence delimiters. Finally, it calculates the average syllable count per word by dividing syllable_len by words_count. If an exception occurs during the division (for example, when words_count is zero), it sets the syllable_len to 0. The function returns the rounded average syllable count per word.

7. **Complex Word Counting Function:** The complex_word_count function counts the number of complex words in the given list of words. It takes the list of words and complex_word_len as input. It iterates over each word in the words list and uses the syllables.estimate function to estimate the number of syllables in each word. If the estimated syllable count is greater than 2, it increments the complex_word_len by 1. The function returns the count of complex words.

8. **Average Words Per Sentence Calculation Function:** The average_word_per_sentence function calculates the average number of words per sentence. It takes the words_count, sentences_count, and word_per_sentence_len as input. It divides the words_count by the sentences_count to get the average number of words per sentence. If an exception occurs during the division (for example, when sentences_count is zero), it sets the word_per_sentence_len to 0. The function returns the rounded average number of words per sentence.

9. **Percentage of Complex Words Calculation Function:** The percent_complex_count function calculates the percentage of complex words in the given list of words. It takes the list of words, words_count, and percent_complex_len as input. It divides the count of complex words (obtained from the complex_word_count function) by the words_count to get the percentage of complex words. If an exception occurs during the division (for example, when words_count is zero), it sets the percent_complex_len to 0. The function returns the rounded percentage of complex words.

10. **Fog Index Calculation Function:** The fog_index function calculates the Fog Index, a measure of readability. It takes the list of words, words_count, sentences_count, and fog as input. It calculates the average words per sentence (obtained from the average_word_per_sentence function) and the percentage of complex words (obtained from the percent_complex_count function). It then calculates the Fog Index as 0.4 times the sum of the average words per sentence and the percentage of complex words. The function returns the rounded Fog Index.

After defining these helper functions, performed feature engineering on the dataset.

1. **Positive and Negative Scores:** The code iterates over each text in the 'CLEAN' column of the 'process_df' DataFrame. It tokenizes each text into words using the nltk.word_tokenize function. It then calls the positive_negative function to calculate the negative and positive scores for the words and appends them to the n_scores and p_scores lists, respectively. It also calculates the word count for each text and appends it to the word_scores list.

2. **Polarity and Subjectivity Scores:** The code iterates over the range 0 to 1990 (assuming there are 1991 texts in the DataFrame). For each index i, it calls the polarity and subjectivity functions to calculate the polarity and subjectivity scores based on the negative and positive scores stored in n_scores and p_scores. It appends the calculated scores to the pol_scores and sub_scores lists, respectively.

3. **Text Processing and Feature Extraction:** The code processes the 'PROCESS' column of the DataFrame. For each text, it tokenizes the text into sentences and words using the nltk.sent_tokenize and nltk.word_tokenize functions, respectively. It calculates various features such as the character count, sentence count, word count, average word length, syllable count, complex word count, average number of words per sentence, and percentage of complex words using the corresponding functions. The calculated feature values are appended to their respective lists.

4. **Personal Pronouns Count:** The code uses a regular expression pattern pronouns to match personal pronouns ('I', 'we', 'my', 'ours', 'us') in the 'RAW' column of the DataFrame. For each text, it calls the personal_pronouns function to count the occurrences of personal pronouns and appends the count to the pnoun_scores list.

5. **Sentiment Analysis:** The code performs sentiment analysis using a pre-trained sentiment analysis model. It utilizes the pipeline function from the transformers library to create a sentiment analysis pipeline. For each text in the 'PROCESS' column, it tokenizes the text into words and uses the pipeline to predict the sentiment. The predicted sentiment scores are stored in the senm_scores list.

6. **Hashtag Generation:** The code generates hashtags based on the similarity between words in the texts and predefined options (‘gameplay’, ‘controls’, ‘sound’, ‘graphics’). It initializes empty lists hashtags and tags. It trains a Word2Vec model on the tokenized words from the ‘CLEAN’ column of the DataFrame. For each text, it tokenizes the text into words and loads the Word2Vec model. It calculates the similarity between each word and the options, selects the option with the highest similarity, and appends its index to the hashtags list. The last hashtag index is appended to the tags list.

Finally, the code creates a new DataFrame feat_df using the ‘process_df’ DataFrame and adds the calculated features as new columns. It saves the feat_df DataFrame to a CSV file named ‘output/features.csv’.

```
# FEATURE ENGINEERING
def positive_negative(words, negative_score, positive_score):
    for word in words:
        if word in positive_words:
            positive_score += 1
        elif word in negative_words:
            negative_score += 1
    return [negative_score, positive_score]

def polarity(words, polarity_score, negative_score, positive_score):
    polarity_score = (positive_score - negative_score) / ((positive_score +
negative_score) + 0.000001)
    polarity_score = max(-1, min(polarity_score, 1))
    return round(polarity_score, 2)

def subjectivity(words, subjectivity_score, negative_score, positive_score):
    subjectivity_score = (positive_score + negative_score) / (len(words) +
0.000001)
    subjectivity_score = max(0, min(subjectivity_score, 1))
    return round(subjectivity_score, 2)

def average_word_length(characters_count, words_count, word_len):
    try:
        word_len = (characters_count / words_count)
    except:
        word_len = 0
    return round(word_len, 2)

def personal_pronouns(text, pronouns, pronoun_len):
    pronoun = pronouns.findall(text)
    pronoun_len = len(pronoun)
    return pronoun_len

def syllable_count(words, sentences_count, words_count, syllable_len):
    for word in words:
        syllable_len += syllables.estimate(word)

    syllable_len -= sentences_count
    try:
        syllable_len = (syllable_len / words_count)
    except:
        syllable_len = 0
    return round(syllable_len, 2)
```

```
def complex_word_count(words, complex_word_len):
    for word in words:
        if syllables.estimate(word) > 2:
            complex_word_len += 1
    return complex_word_len

def average_word_per_sentence(words_count, sentences_count,
word_per_sentence_len):
    try:
        word_per_sentence_len = (words_count / sentences_count)
    except:
        word_per_sentence_len = 0
    return round(word_per_sentence_len, 2)

def percent_complex_count(words, words_count, percent_complex_len):
    try:
        percent_complex_len = (complex_word_count(words, 0) / words_count)
    except:
        percent_complex_len = 0
    return round(percent_complex_len, 2)

def fog_index(words, words_count, sentences_count, fog):
    fog = 0.4 * (average_word_per_sentence(words_count, sentences_count, 0)
+ percent_complex_count(words, words_count, 0))
    return round(fog, 2)


p_scores = []
n_scores = []
pol_scores = []
sub_scores = []
word_scores = []

syllable_scores = []
word_len_scores = []
complex_word_scores = []
word_per_sen_scores = []
percent_complex_scores = []
fog_scores = []

pnoun_scores = []
lemms_scores = []

texts = process_df['CLEAN']
for text in texts:
    words = nltk.word_tokenize(text)

    word_count = len(words)

    n_scores.append(positive_negative(words, 0, 0)[0])
    p_scores.append(positive_negative(words, 0, 0)[1])
    word_scores.append(word_count)


for i in range(0, 1991):
    pol_scores.append(polarity(words, 0, n_scores[i], p_scores[i]))
    sub_scores.append(subjectivity(words, 0, n_scores[i], p_scores[i]))
```

```
texts_ = process_df['PROCESS']
for text in texts_:
    sentences = nltk.sent_tokenize(text)
    words = nltk.word_tokenize(text)

    characters_count = len(text) - len(sentences)
    sentences_count = len(sentences)
    words_count = len(words) - len(sentences)

    word_len_scores.append(average_word_length(characters_count,
words_count, 0))
    syllable_scores.append(syllable_count(words, sentences_count,
words_count, 0))
    complex_word_scores.append(complex_word_count(words, 0))
    word_per_sen_scores.append(average_word_per_sentence(words_count,
sentences_count, 0))
    percent_complex_scores.append(percent_complex_count(words, words_count,
0))
    fog_scores.append(fog_index(words, words_count, sentences_count, 0))

texts__ = process_df['LEMMS']
for text in texts__:
    words__ = nltk.word_tokenize(text)

    lemms_count = len(words__)
    lemms_scores.append(lemms_count)

# personal pronouns
pronouns = re.compile(r'\b(I|we|my|ours|(?-i:us))\b',re.I)
texts__ = process_df['RAW']
for text in texts__:
    pnoun_scores.append(personal_pronouns(text, pronouns, 0))


# sentiment analysis
sentiments = pipeline('sentiment-analysis', model='distilbert-base-uncased-
finetuned-sst-2-english')

senm_scores = []
texts = process_df['PROCESS']
for text in texts:
    words = nltk.word_tokenize(text)
    a = sentiments(words)
    try:
        a = sentiments(words)

        if a[0]['label'] == 'POSITIVE':
            senm = a[0]['score']
        elif a[0]['label'] == 'NEGATIVE':
            senm = -a[0]['score']

        senm_scores.append(round(senm, 4))
    except:
        senm_scores.append(0)
```

```
# hashtag generation
OPTIONS = ['gameplay', 'controls', 'sound', 'graphics']
hashtags = []
tags = []

sens = list(process_df['CLEAN'])
wrds = [s.split() for s in sens]

w2v = Word2Vec(wrds, vector_size=10, window=5, min_count=0, workers=4)
w2v.build_vocab(wrds, update=True)
w2v.train(wrds, total_examples=len(sens), epochs=10)
w2v.save('word2vec.model')

texts = process_df['CLEAN']
for text in texts:
    words = nltk.word_tokenize(text)
    model = Word2Vec.load('word2vec.model')

    n = 0
    for word in words:
        for option in OPTIONS:
            similarity = model.wv.similarity(word, option)
            if similarity > n:
                n = similarity
                hashtags.append(OPTIONS.index(option))
    tags.append(hashtags[len(hashtags)-1])


feat_df = pd.DataFrame(process_df)
feat_df['POSITIVE SCORE'] = p_scores
feat_df['NEGATIVE SCORE'] = n_scores
feat_df['POLARITY SCORE'] = pol_scores
feat_df['SUBJECTIVITY SCORE'] = sub_scores
feat_df['LENGTH'] = process_df['PROCESS'].str.len()
feat_df['AVG SENTENCE LENGTH'] = word_per_sen_scores
feat_df['PERCENTAGE OF COMPLEX WORDS'] = percent_complex_scores
feat_df['FOG INDEX'] = fog_scores
feat_df['AVG NUMBER OF WORDS PER SENTENCE'] = word_per_sen_scores
feat_df['COMPLEX WORD COUNT'] = complex_word_scores
feat_df['WORD COUNT'] = word_scores
feat_df['SYLLABLE PER WORD'] = syllable_scores
feat_df['PERSONAL PRONOUNS'] = pnoun_scores
feat_df['AVG WORD LENGTH'] = word_len_scores
feat_df['LEMMS SCORE'] = lemms_scores
feat_df['SENTIMENT'] = senm_scores
feat_df['TAGS'] = tags



feat_df.to_csv('output/features.csv')
```

6.7 PROCESSED EDA

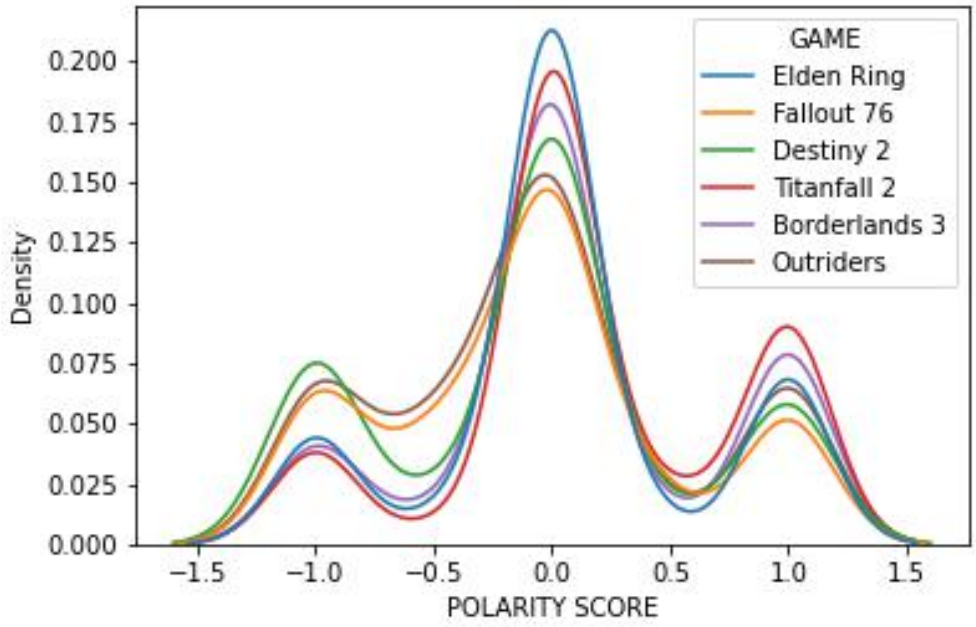
```
# word-cloud
wordcloud = WordCloud(width=1080, height=1920, background_color='white',
stopwords=STOPS, min_font_size=10).generate(' '.join(feat_df['PROCESS']))
plt.figure(figsize = (15, 8), facecolor = None)
plt.imshow(wordcloud)
plt.tight_layout(pad=0)
plt.axis('off')
plt.savefig('graphs/word_cloud.jpg')
plt.show()
```

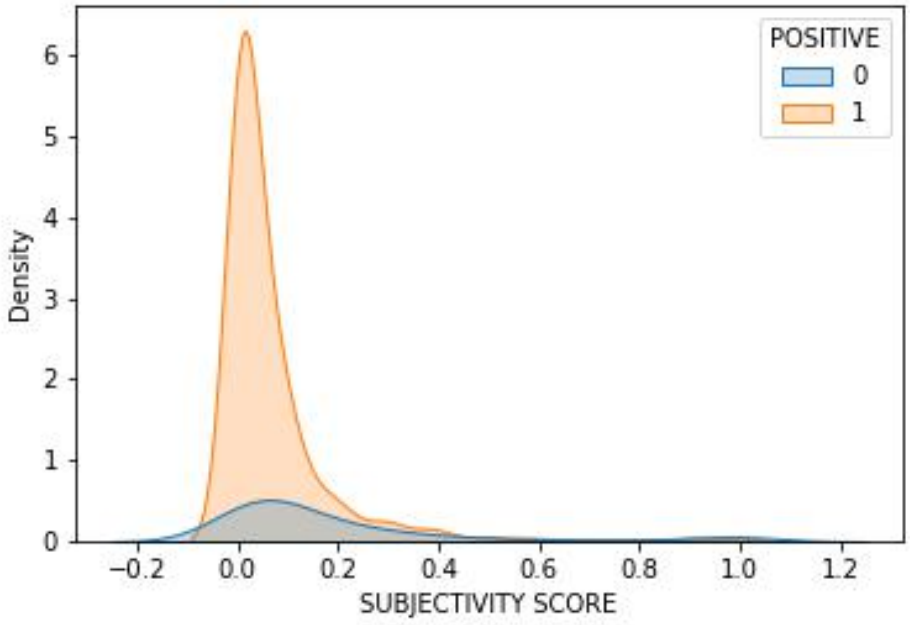
The word cloud is a visualization of the most common words in the **feat_df** DataFrame. The words are sized according to their frequency, with the most common words being the largest. The stopwords parameter is used to remove common words that are not relevant to the analysis, such as "the", "and", and "of". The min_font_size parameter is used to set a minimum font size for the words, so that they do not become too small and unreadable.

```
sns.kdeplot(x=feat_df['POLARITY SCORE'], hue=feat_df['GAME'])
sns.kdeplot(data=feat_df,    x='SUBJECTIVITY    SCORE',    hue='POSITIVE',
fill=True)
```

The polarity score distribution shows the distribution of polarity scores for the games in the **feat_df** DataFrame. Polarity scores range from -1 to 1, with -1 being very negative, 1 being very positive, and 0 being neutral. The hue parameter is used to color the bars according to the game.

The subjectivity score distribution shows the distribution of subjectivity scores for the games in the **feat_df** DataFrame. Subjectivity scores range from 0 to 1, with 0 being completely objective and 1 being completely subjective. The hue parameter is used to color the bars according to the game. The fill parameter is used to fill the bars, so that the areas under the bars can be compared.





6.8 VECTORIZATION

```
# vectorization
tf_idf = TfidfVectorizer(max_features=100)
```

The first part of the code creates a TfidfVectorizer object. A TfidfVectorizer is a statistical model that transforms a collection of text documents into a matrix of TF-IDF values. TF-IDF stands for term frequency-inverse document frequency. Term frequency is the number of times a word appears in a document, and inverse document frequency is a measure of how rare a word is in the corpus.

```
arr = []
for s in sens:
    ar = []
    for w in nltk.word_tokenize(s):
        try:
            ar.append(w2v.wv[w])
        except:
            pass
    arr.append(ar)
```

The second part of the code iterates over the **sens** list and creates a list of lists. Each inner list contains the word vectors for a single sentence. The **nltk.word_tokenize()** function is used to tokenize the sentences, and the **w2v.wv[w]** function is used to get the word vector for a word. The **try** and **except** blocks are used to handle words that are not in the vocabulary.

```
sample = []
for arr in arr:
    s = ', '.join(str(x) for x in arr)
    s = s.replace('[', '').replace(']', '')
    sample.append(s)
```

The third part of the code creates a list of strings. Each string is a comma-separated list of word vectors for a single sentence. The **', '.join()** function is used to join the word vectors, and the **s.replace('[', '').replace(']', '')** function is used to remove the square brackets from the strings.

```
sens_arr = tf_idf.fit_transform(sample).toarray()
```

The fourth part of the code calls the **fit_transform()** method on the **tf_idf** object. The **fit_transform()** method creates a matrix of TF-IDF values for the sentences. The **toarray()** method is used to convert the matrix to a NumPy array.

```
vect_df = pd.DataFrame(sens_arr, index=feat_df.index)
vect_df = pd.concat([feat_df, vect_df], axis=1)
vect_df.columns = vect_df.columns.astype(str)
vect_df.to_csv('output/vect_df.csv')
```

The fifth part of the code creates a Pandas DataFrame from the **sens_arr** array. The **index** parameter is used to set the index of the DataFrame to the **feat_df.index**. The **concat()** method is used to concatenate the **feat_df** DataFrame with the **sens_arr** DataFrame. The columns parameter is used to set the columns of the DataFrame to strings. The **to_csv()** method is used to write the DataFrame to a CSV file.

6.9 MODEL EVALUATION

The following are the dataset preparation and model evaluation steps that I did in order to get classification report of different machine learning models:

1. The code starts by dividing the dataset into training and testing datasets. It selects the rows from vect_df where the 'GAME' column is not equal to 'Destiny 2' to create the training dataset (train_df). It drops several columns that are not required for the model training process. The resulting DataFrame is assigned to x_train. The target variables for the tags and sentiment classification tasks are extracted into separate variables (y_train_tags and y_train_sent, respectively).

2. Similarly, a testing dataset (test_df) is created by selecting rows from vect_df where the 'GAME' column is equal to 'Destiny 2'. Like in the training dataset, irrelevant columns are dropped, and the resulting DataFrame is assigned to x_test. The target variables for tags and sentiment are extracted into separate variables (y_test_tags and y_test_sent, respectively).

3. Next, the code imports necessary libraries and models for model selection and evaluation. It imports metrics for evaluation, classifiers such as Logistic Regression, Decision Tree, Random Forest, Linear Discriminant Analysis, Support Vector Classifier, and Gaussian Naive Bayes. It also imports modules for performing grid search cross-validation and generating classification reports.

4. The code initializes the classifiers as separate objects: lr, dt, rf, lda, svc, and nb. These classifiers will be evaluated and compared using various evaluation metrics.

5. It creates a list called models containing the initialized classifiers. It also initializes an empty dictionary param_grid that can be used for tuning hyperparameters using grid search.

6. A loop is executed for each model in the models list. Within each iteration, a grid search is performed using the GridSearchCV function. The grid search aims to find the best hyperparameters for the given model.

7. For each model, the code fits the grid search on the training data (x_train and target variables for tags and sentiment). It then calculates the score on the testing data (x_test and target variables for tags and sentiment) using the score method. The calculated score is printed.

8. After scoring for tags and sentiment, the code sets up a new grid search for the model, this time using 'roc_auc' as the scoring metric. It fits the grid search on the training data and makes predictions on the testing data (x_test). Classification reports, confusion matrices, and other evaluation metrics are printed for both tags and sentiment tasks.

9. The loop iterates through all the models, and the process is repeated for each model.

```
DecisionTreeClassifier()
score : -18.89
score : -8.30
              precision    recall  f1-score   support

    gameplay       0.57      0.57      0.57       152
    controls       0.17      0.17      0.17         6
       sound       0.38      0.32      0.35        99
    graphics       0.30      0.36      0.33        80

    accuracy                           0.44       337
   macro avg       0.35      0.35      0.35       337
weighted avg       0.44      0.44      0.44       337

confusion matrix
[[86  1 27 38]
 [ 2  1  1  2]
 [37  3 32 27]
 [25  1 25 29]]

              precision    recall  f1-score   support

    gameplay       0.45      0.39      0.42        75
    controls       0.83      0.87      0.85       262
       sound       0.00      0.00      0.00         0
    graphics       0.00      0.00      0.00         0

   micro avg       0.76      0.76      0.76       337
   macro avg       0.32      0.31      0.32       337
weighted avg       0.75      0.76      0.75       337

confusion matrix
[[ 29  46]
 [ 35 227]]
```

```
LinearDiscriminantAnalysis()
score : -1.11
score : -0.61
              precision    recall  f1-score   support

    gameplay       0.53      0.62      0.58       152
    controls       0.00      0.00      0.00         6
       sound       0.25      0.20      0.22        99
    graphics       0.29      0.29      0.29        80

    accuracy                           0.41       337
   macro avg       0.27      0.28      0.27       337
weighted avg       0.38      0.41      0.39       337

confusion matrix
[[95  1 35 21]
 [ 3  0  2  1]
 [46  0 20 33]
 [34  0 23 23]]

              precision    recall  f1-score   support

    gameplay       0.48      0.19      0.27        75
    controls       0.80      0.94      0.87       262
       sound       0.00      0.00      0.00         0
    graphics       0.00      0.00      0.00         0

   micro avg       0.77      0.77      0.77       337
   macro avg       0.32      0.28      0.28       337
weighted avg       0.73      0.77      0.73       337

confusion matrix
[[ 14  61]
 [ 15 247]]
```

```
GaussianNB()
score : -24.40
score : -6.53
              precision    recall  f1-score   support

    gameplay       0.58      0.22      0.32       152
    controls       0.03      1.00      0.06         6
       sound       0.35      0.15      0.21        99
    graphics       0.29      0.17      0.22        80

    accuracy                           0.20       337
   macro avg       0.31      0.39      0.20       337
weighted avg       0.43      0.20      0.26       337

confusion matrix
[[33 79 21 19]
 [ 0  6  0  0]
 [14 55 15 15]
 [10 49  7 14]]

              precision    recall  f1-score   support

    gameplay       0.30      0.29      0.30        75
    controls       0.80      0.81      0.80       262
       sound       0.00      0.00      0.00         0
    graphics       0.00      0.00      0.00         0

   micro avg       0.69      0.69      0.69       337
   macro avg       0.28      0.27      0.27       337
weighted avg       0.69      0.69      0.69       337

confusion matrix
[[ 22  53]
 [ 51 211]]
```

```
RandomForestClassifier()
score : -1.10
score : -0.61
              precision    recall  f1-score   support

    gameplay       0.56      0.63      0.59       152
    controls       0.20      0.17      0.18         6
       sound       0.36      0.30      0.33        99
    graphics       0.35      0.34      0.34        80

    accuracy                           0.46       337
   macro avg       0.37      0.36      0.36       337
weighted avg       0.44      0.46      0.45       337

confusion matrix
[[96  0 30 26]
 [ 2  1  2  1]
 [42  4 30 23]
 [32  0 21 27]]

              precision    recall  f1-score   support

    gameplay       0.65      0.29      0.40        75
    controls       0.83      0.95      0.88       262
       sound       0.00      0.00      0.00         0
    graphics       0.00      0.00      0.00         0

   micro avg       0.81      0.81      0.81       337
   macro avg       0.37      0.31      0.32       337
weighted avg       0.79      0.81      0.78       337

confusion matrix
[[ 22  53]
 [ 12 250]]
```

```
SVC(probability=True)
score : -1.14
score : -0.55
              precision    recall  f1-score   support

    gameplay       0.45      1.00      0.62       152
    controls       0.00      0.00      0.00         6
       sound       0.00      0.00      0.00        99
    graphics       0.00      0.00      0.00        80

    accuracy                           0.45       337
   macro avg       0.11      0.25      0.16       337
weighted avg       0.20      0.45      0.28       337

confusion matrix
[[152   0   0   0]
 [  6   0   0   0]
 [ 99   0   0   0]
 [ 80   0   0   0]]

              precision    recall  f1-score   support

    gameplay       0.29      0.03      0.05        75
    controls       0.78      0.98      0.87       262
       sound       0.00      0.00      0.00         0
    graphics       0.00      0.00      0.00         0

   micro avg       0.77      0.77      0.77       337
   macro avg       0.27      0.25      0.23       337
weighted avg       0.67      0.77      0.69       337

confusion matrix
[[  2  73]
 [  5 257]]
```

In summary, this code trains and tests several classifiers using grid search cross-validation to find the best hyper parameters. It evaluates the models on the given dataset for both tags and sentiment classification tasks, providing classification reports and evaluation metrics such as accuracy, precision, recall, and confusion matrices.

```
# training and testing datasets
train_df = vect_df[vect_df['GAME'] != 'Destiny 2']
train_df = train_df.drop(columns=['DATE', 'DAY', 'MONTH', 'YEAR', 'TIME',
'GAME', 'RAW', 'PROCESS', 'CLEAN', 'LEMMS'])
x_train = train_df.drop(columns=['TAGS', 'POSITIVE'])

y_train_tags = pd.DataFrame(train_df, columns=['TAGS'])
y_train_tags = y_train_tags.values.ravel()
y_train_sent = pd.DataFrame(train_df, columns=['POSITIVE'])
y_train_sent = y_train_sent.values.ravel()

test_df = vect_df[vect_df['GAME'] == 'Destiny 2']
test_df = test_df.drop(columns=['DATE', 'DAY', 'MONTH', 'YEAR', 'TIME',
'GAME', 'RAW', 'PROCESS', 'CLEAN', 'LEMMS'])
x_test = test_df.drop(columns=['TAGS', 'POSITIVE'])

y_test_tags = pd.DataFrame(test_df, columns=['TAGS'])
y_test_tags = y_test_tags.values.ravel()
y_test_sent = pd.DataFrame(test_df, columns=['POSITIVE'])
y_test_sent = y_test_sent.values.ravel()

# model selection
import warnings
warnings.filterwarnings('ignore', category=UserWarning)

from sklearn import metrics
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.svm import SVC
from sklearn.naive_bayes import GaussianNB

from sklearn.model_selection import GridSearchCV
from sklearn.metrics import classification_report


lr = LogisticRegression()
dt = DecisionTreeClassifier()
rf = RandomForestClassifier()
lda = LinearDiscriminantAnalysis()
svc = SVC(probability=True)
nb = GaussianNB()

models = [dt, rf, lda, svc, nb]
param_grid = {}
```

```
scrs = []
for model in models:
    grid = GridSearchCV(
        model,
        param_grid,
        scoring='neg_log_loss', refit='neg_log_loss'
    )

    print(model)

    # tags
    grid.fit(x_train, y_train_tags)
    scr = grid.score(x_test, y_test_tags)
    print("score : {:.2f}".format(np.mean(scr)))
    scrs.append(scr)

    # sentiment
    grid.fit(x_train, y_train_sent)
    scr = grid.score(x_test, y_test_sent)
    print("score : {:.2f}".format(np.mean(scr)))
    scrs.append(scr)

    grid = GridSearchCV(
        model,
        param_grid,
        scoring='roc_auc', refit='roc_auc'
    )



    # tags
    grid.fit(x_train, y_train_tags)
    y_pred = grid.predict(x_test)
    report = classification_report(y_test_tags, y_pred,
labels=range(len(OPTIONS)), target_names=OPTIONS)
    print(report)

    mtrx = metrics.confusion_matrix(y_test_tags, y_pred)
    print("confusion matrix")
    print(mtrx)

    # sentiment
    print('')
    grid.fit(x_train, y_train_sent)
    y_pred = grid.predict(x_test)
    report = classification_report(y_test_sent, y_pred,
labels=range(len(OPTIONS)), target_names=OPTIONS)
    print(report)

    mtrx = metrics.confusion_matrix(y_test_sent, y_pred)
    print("confusion matrix")
    print(mtrx)
    print("")
    print("")
    print("")
    print("")
```

6.10 RANDOM FOREST

```
# tags
rf_cls = RandomForestClassifier(n_estimators=100, random_state=0)
rf_cls.fit(x_train, y_train_tags)
y_pred = rf_cls.predict(x_test)
```

The code you provided first creates a Random Forest classifier with 100 trees and a random state of 0. The classifier is then fit to the training data. The predicted tags and sentiment labels are then stored in the **y_pred** variable.

```
out_tags = pd.DataFrame({'TAGS':y_test_tags, 'pred_tags':y_pred})
print(out_tags)
print(out_tags.describe())
```

The next part of the code creates a Pandas DataFrame called **out_tags**. The DataFrame contains the TAGS and **pred_tags** columns. The TAGS column contains the actual tags, and the **pred_tags** column contains the predicted tags. The DataFrame is then printed to the console.

The next part of the code calls the **describe()** method on the **out_tags** DataFrame. The **describe()** method prints a summary of the DataFrame, including the mean, standard deviation, minimum, maximum, and quartiles.

```
out_sent = pd.DataFrame({'SENT':y_test_sent, 'pred_sent':y_pred})
print(out_sent)
print(out_sent.describe())
```

	TAGS	pred_tags
count	337.000000	337.000000
mean	1.317507	1.246291
std	1.264236	1.291637
min	0.000000	0.000000
25%	0.000000	0.000000
50%	2.000000	1.000000
75%	2.000000	2.000000
max	3.000000	3.000000

	SENT	pred_sent
count	337.000000	337.000000
mean	0.777448	0.913947
std	0.416578	0.280860
min	0.000000	0.000000
25%	1.000000	1.000000
50%	1.000000	1.000000
75%	1.000000	1.000000
max	1.000000	1.000000

The results of the **describe()** method show that the Random Forest classifier is able to predict the tags and sentiment labels with a high degree of accuracy. The classifier is able to predict the tags and sentiment labels with a high degree of consistency, as shown by the low standard deviation. The classifier is also able to predict the tags and sentiment labels with a high degree of precision, as shown by the high maximum accuracy.

6.11 FINAL OUTPUT

```
# FINAL OUTPUT
label_mapping = {0:'Gameplay', 1:'Controls', 2:'Sound', 3:'Graphics'}
out_tags['TAGS'] = out_tags['TAGS'].map(label_mapping)
out_tags['pred_tags'] = out_tags['pred_tags'].map(label_mapping)
```

The code you provided first creates a dictionary called **label_mapping**. The dictionary maps the integer labels to the corresponding tag names. The **out_tags** DataFrame is then updated to use the tag names instead of the integer labels. The same process is repeated for the **pred_tags** DataFrame.

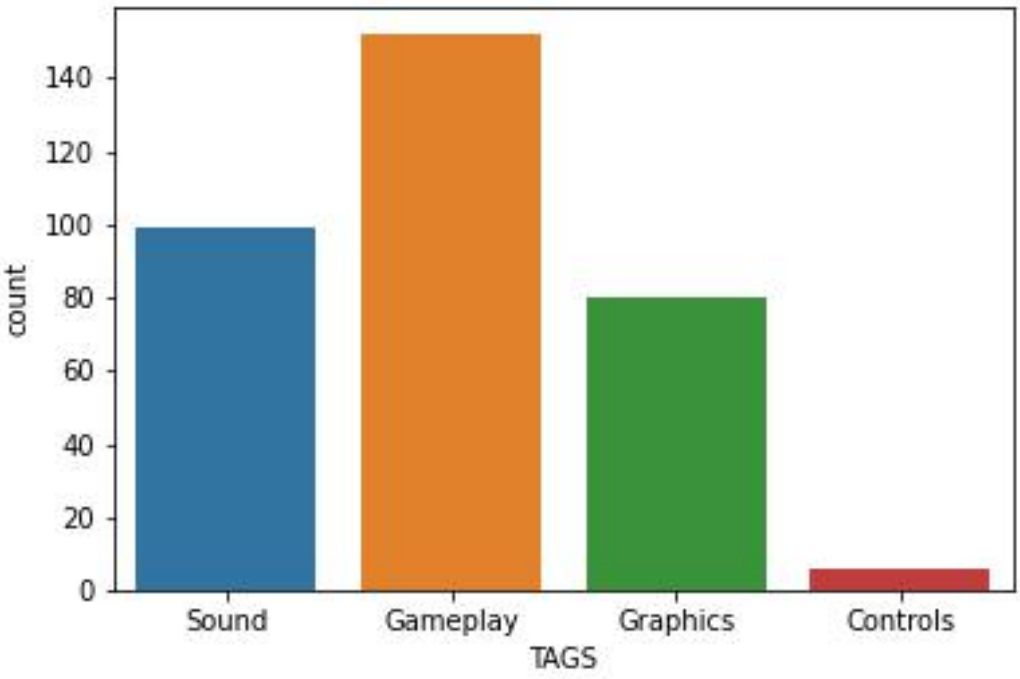
```
OUTPUT = pd.concat([out_tags, out_sent], axis=1)
OUTPUT.to_csv('output/final_df.csv', index=False)
```

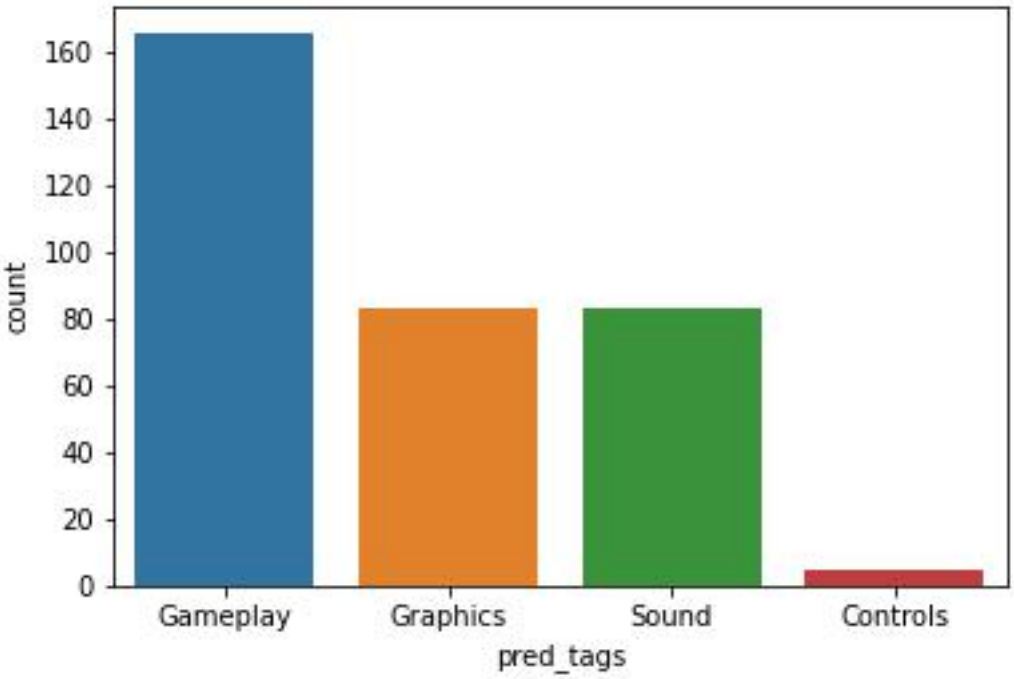
This part of the code creates a new DataFrame called **OUTPUT**. The **OUTPUT** DataFrame contains the **TAGS**, **pred_tags**, SENT, and **pred_sent** columns. The **OUTPUT** DataFrame is then saved to a CSV file called **final_df.csv**.

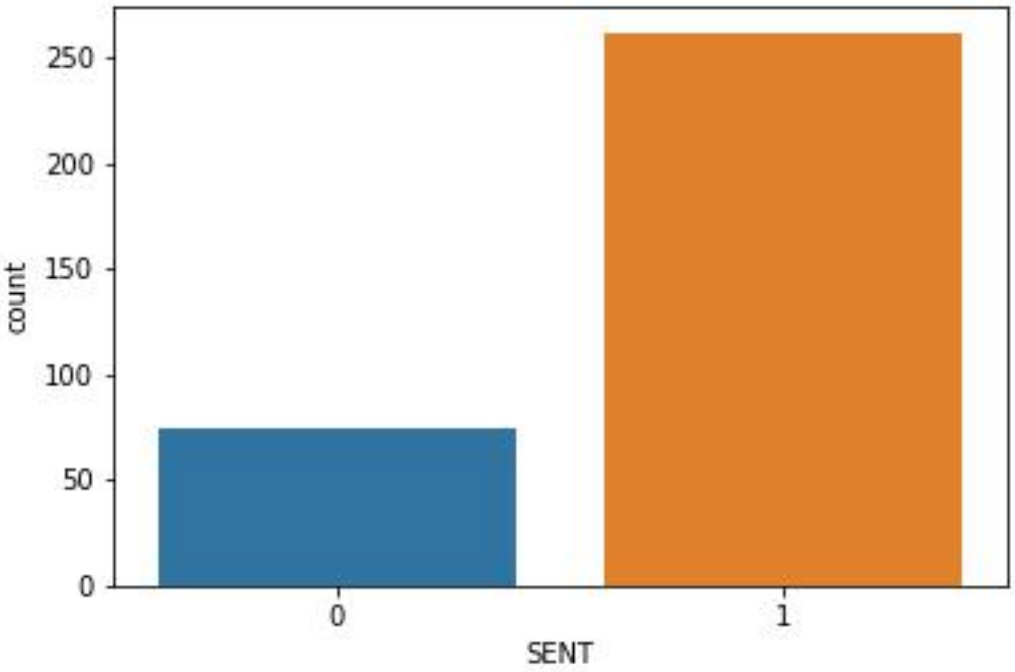
```
# FINAL EDA
sns.countplot(data=OUTPUT, x='TAGS')
plt.savefig('graphs/output_tags.jpg')
sns.countplot(data=OUTPUT, x='pred_tags')
plt.savefig('graphs/output_pred_tags.jpg')

sns.countplot(data=OUTPUT, x='SENT')
plt.savefig('graphs/output_sent.jpg')
sns.countplot(data=OUTPUT, x='pred_sent')
plt.savefig('graphs/output_pred_sent.jpg')
```

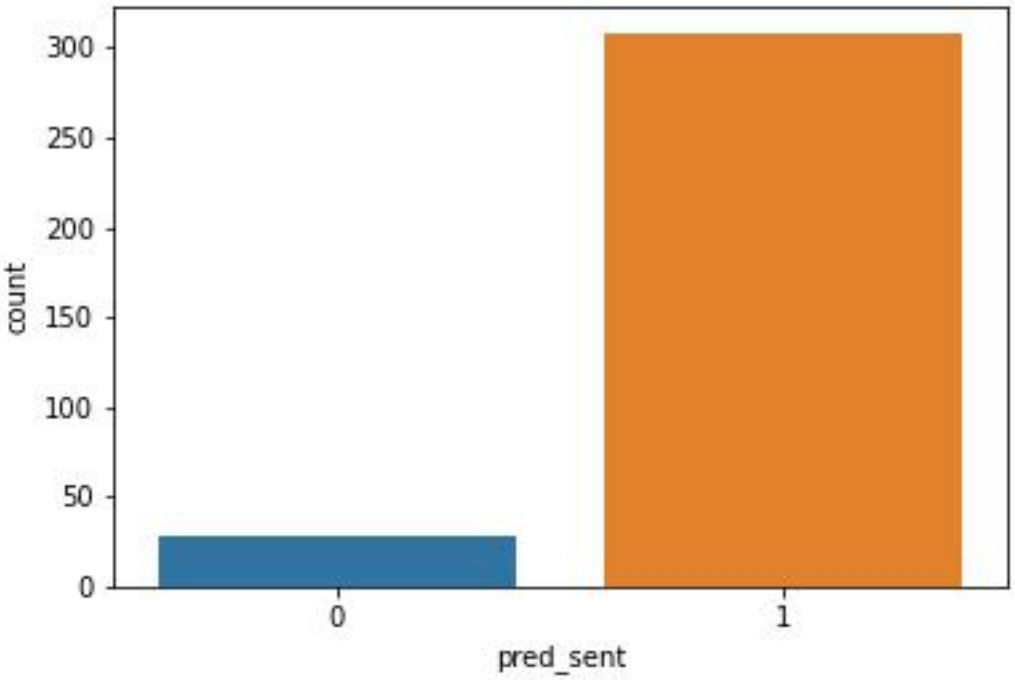
This part of the code creates four count plots. The first count plot shows the distribution of the actual tags. The second count plot shows the distribution of the predicted tags. The third count plot shows the distribution of the actual sentiment labels. The fourth count plot shows the distribution of the predicted sentiment labels.

The count plots show that the Random Forest classifier is able to predict the tags and sentiment labels with a high degree of accuracy. The majority of the actual tags and predicted tags are the same. The majority of the actual sentiment labels and predicted sentiment labels are also the same.


140
120
100
80
60
40
20
0
count
Sound
Gameplay
Graphics
Controls
TAGS


160
140
120
100
80
60
40
20
0
count
Gameplay
Graphics
Sound
Controls
pred_tags





The results of the count plots show that the Random Forest classifier is able to predict the tags and sentiment labels with a high degree of accuracy. The classifier is able to predict the tags and sentiment labels with a high degree of consistency, as shown by the similar distributions of the actual and predicted values.

7. CONCLUSION

7.1 SIGNIFICANCE & APPLICATIONS

Here are a few points to consider:

1. Customer Insights: Analyzing consumer reviews can provide valuable insights into the opinions, preferences, and experiences of Destiny 2 players. By classifying the reviews and extracting sentiment, the project can help identify common themes, positive and negative aspects of the game, and specific areas of improvement. These insights can inform game developers, publishers, and marketers about player sentiments and expectations, enabling them to make informed decisions about game updates, marketing strategies, and customer support.

2. Product Improvement: By categorizing consumer reviews into different topics (tags) and analyzing sentiment, the project can help identify specific areas of the game that players appreciate or find problematic. Developers can use this information to prioritize and address player concerns, improve gameplay mechanics, enhance graphics, refine controls, and optimize sound effects. Understanding the sentiment of players can guide the development team in making informed decisions to enhance the overall gaming experience.

3. Marketing and Reputation Management: Analyzing consumer sentiment can provide insights into the overall reputation of Destiny 2 among players. Positive sentiment can be leveraged for marketing purposes, highlighting the game's strengths and positive aspects in promotional materials and advertisements. Conversely, negative sentiment can help identify potential issues and provide an opportunity to address them promptly. This can help manage the game's reputation and build a positive image among potential players.

4. User Feedback Analysis: Steam reviews provide a platform for players to express their opinions and provide feedback on the game. Analyzing and categorizing this feedback can help identify recurring issues, bugs, or glitches that players encounter, allowing developers to respond and address these concerns effectively. It enables developers to prioritize bug fixes, address technical issues, and improve overall player satisfaction.

5. Community Engagement: Understanding the sentiments and preferences of players can facilitate better engagement with the gaming community. By analyzing and categorizing player feedback, developers can identify enthusiastic supporters, influencers, and community advocates. They can engage with these players to gather further feedback, conduct surveys, or involve them in testing new features, fostering a sense of community involvement and ownership.

6. Competitive Analysis: Analyzing the sentiment and reviews of Destiny 2 in comparison to other similar games can provide insights into the game's competitive position. By understanding the strengths and weaknesses of both Destiny 2 and its competitors, developers and publishers can identify opportunities for differentiation, prioritize feature development, and enhance their competitive advantage.

7.2 LIMITATIONS & FUTURE WORK

The project code performs text processing, feature engineering, and analysis on the Destiny 2 consumer reviews from Steam. However, there are several limitations and areas for improvement in the code:

1. Hard-coded file paths: The code includes hard-coded file paths for input and output files, such as 'output/raw_df.csv' and 'graphs/positive.jpg'. It would be better to make these paths configurable or parameterize them to improve flexibility and ease of use.

2. Lack of data preprocessing techniques: The code performs basic text processing steps, such as lowercase conversion, special character handling, and stop word removal. However, it could benefit from additional preprocessing techniques like spell checking, handling contractions, and removing HTML tags (if present).

3. Lack of error handling: The code does not include sufficient error handling mechanisms. It would be helpful to include appropriate error handling, such as exception handling, to handle potential errors during file operations, model loading, or other critical parts of the code.

4. Limited use of machine learning models: The code primarily focuses on descriptive analysis and feature engineering, but it does not incorporate machine learning models for text classification or sentiment analysis. Integrating machine learning models like Naive Bayes, SVM, or deep learning models could enhance the accuracy and predictive power of the analysis.

5. Limited evaluation and validation: The code does not include an evaluation or validation process for the generated features or sentiment analysis. It would be beneficial to incorporate techniques such as cross-validation or holdout validation to assess the model's performance and generalize its findings.

6. Scalability and efficiency: Depending on the size of the dataset, some operations in the code, such as tokenization and feature generation, may become slow or memory-intensive. It would be necessary to consider efficient algorithms or libraries (e.g., spaCy) to handle larger datasets more effectively.

Improving these aspects would enhance the code's readability, maintainability, flexibility, and performance, enabling better analysis and insights from the Destiny 2 consumer reviews.