NLP Project: Movie Reviews Extraction and Analysis

Youssef Abdelhedi, Aicha Ettriki and Khalil Habassi

February 13, 2025

Contents

1 Introduction

This project demonstrates how to extract, clean, analyze, and model movie reviews from websites such as Letterboxd and Rotten Tomatoes. Each step of the process, including code and explanation, is detailed below.

2 Data Extraction from Letterboxd

Objective: Scrape movie reviews, ratings, and details such as titles, genres, duration and producers from Letterboxd.
Letterboxd is a dynamic website where content, such as movie titles, is generated and rendered by JavaScript. This means that traditional scraping techniques, like using `requests` and `BeautifulSoup`, are insufficient because they cannot interact with dynamically rendered content. To overcome this limitation, we use **Selenium**, a tool that allows controlling a web browser to interact with pages as if a human user were navigating them.

2.1 Extraction Code with Selenium

The following code demonstrates how to use Selenium to extract movie titles from multiple pages on Letterboxd.
Code:

```python

from selenium import webdriver
from selenium.webdriver.chrome.service import Service
from selenium.webdriver.chrome.options import Options
from selenium.webdriver.common.by import By
import time
import pandas as pd

driver_path = r"C:\Users\Desktop\M2_BDIA\NLP\Projet_movie\chromedriver-
    win64\chromedriver-win64\chromedriver.exe"
```

Listing 1: Import necessary libraries and the driver path

```python
# Create a Service object
service = Service(driver_path)

# Initialize Chrome browser options
options = Options()
options.add_argument("--headless")  # Run Chrome in headless mode
options.add_argument("--no-sandbox")
options.add_argument("--disable-dev-shm-usage")

# Start Chrome browser with the service and options
driver = webdriver.Chrome(service=service, options=options)

# Access the page
base_url = "https://letterboxd.com/films/popular/this/week/page/"
page_number = 1
film_titles = []

# Function to extract movies from a page
```

```
def get_films():
    film_divs = driver.find_elements(By.CSS_SELECTOR, "div.react-
        component.poster.film-poster")
    for div in film_divs:
        title = div.get_attribute("data-film-name")
        if title:
            film_titles.append(title)

# Scrape movies across multiple pages
clicks = 0
max_clicks = 50  # Scrape 50 pages
while clicks < max_clicks:
    url = f"{base_url}{page_number}/"  # Generate URL for each page
    driver.get(url)
    time.sleep(5)  # Wait for the page to load

    get_films()  # Extract movies from the current page
    print(f"Page {page_number} scraped.")

    page_number += 1  # Move to the next page
    clicks += 1

# Close the browser
driver.quit()
```

Listing 2: Scraping Movie Titles Using Selenium

Explanation:

- The code uses Selenium to open a headless Chrome browser, simulating a user visiting Letterboxd's "Popular Movies" page. It dynamically loads the movies' data using JavaScript, and Selenium interacts with these elements to scrape information.
- The get films() function extracts movie titles from each page by locating elements on the page (using CSS selectors) that hold movie information. It iterates through multiple pages, appending the titles of the movies to a list.
- After scraping the movie titles from 50 pages, the browser session is closed, and the data (3600 films extracted) is saved into a CSV file for further analysis and use.

2.2 Transforming Movie Names and Creating URLs

In order to scrape detailed information from each individual movie on Letterboxd, we first need to generate a valid URL for each movie. The URLs on Letterboxd are structured based on the movie titles, but the titles must be formatted correctly by removing special characters, spaces, and accents.
We define a function `transformer_nom(film)` to clean and format the movie titles. This function replaces spaces with hyphens, removes special characters, and handles accented characters to ensure the title is URL-friendly.

```
import re
from unidecode import unidecode

def transformer_nom(film):
    film = film.replace(" ", "-")
    film = film.replace("   ", "-")
    film = film.replace("&", "")
    film = film.replace("'", "")
    film = film.replace(".", "-")
    film = film.replace(",", "")
    film = film.replace("#", "").replace(":", "")
    film = film.replace("  ", "e").replace("  ", "e").replace("  ", "a")
    film = re.sub(r'-+', '-', film)
    film = film.replace("***", "")
    film = film.replace("*", "")
    film = film.replace("(", "").replace(")", "")
    film = film.replace("!", "").replace("?", "").replace(":", "").
        replace(";", "").replace('"', "")
    film = re.sub(r'[^\w\s-]', '', film)
    film = film.replace('[', '').replace(']', '')
    film = unidecode(film)
    return film.lower()

df['MOVIE_NAME'] = df['Film Title'].apply(transformer_nom)
```

Listing 3: Fonction to transform movies names

Now that we have the cleaned movie names, we can generate the appropriate URL for each movie. The URL is structured by appending the transformed movie name to the base URL for Letterboxd.

```
base_url = "https://letterboxd.com/film/"
df['URL_FILM'] = base_url + df['MOVIE_NAME'] + "/"
```

Listing 4: The right URL

2.3 Extracting Informations

To gather the necessary information for each column in our dataset, we created individual functions for each piece of information. This approach was necessary because the tags and URLs vary for each type of information we wanted to extract. For example, the reviews section on Letterboxd requires a different approach in terms of URL structure and HTML parsing compared to other pieces of information.

Below is an example of how we extract reviews for each movie from its specific page on Letterboxd:

```
import requests
from bs4 import BeautifulSoup

def fetch_reviews(url):
    reviews_url = url + "reviews/by/activity/"
    response = requests.get(reviews_url)

    if response.status_code != 200:
        print(f"Failed to fetch {reviews_url}")
        return []

    soup = BeautifulSoup(response.text, 'html.parser')

    # Extract reviews
    review_elements = soup.select('div.body-text p')
    reviews = [review.text.strip() for review in review_elements]
    return reviews
```

Listing 5: Fonction to fetch the reviews

To ensure that we correctly capture the reviews for each movie, the `fetch_reviews` function is applied to each movie's URL in the DataFrame. If reviews are unavailable, a placeholder message is added instead. Additionally, the results are saved periodically to avoid losing progress.

Here's how the function is applied:

```
# Iterate through the rows and fetch reviews for each movie
for idx, row in df.iloc[::].iterrows():
    # Add reviews if not already present
    if pd.isna(row["REVIEWS"]) or row["REVIEWS"] == "":
        reviews = fetch_reviews(row['URL_FILM'])
        reviews_text = "**".join(reviews) if reviews else "No reviews
            available"
        df.loc[idx, "REVIEWS"] = reviews_text

    # Save results after each iteration to avoid data loss
    df.to_csv(r"C:\Users\Desktop\M2_BDIA\NLP\Projet_movie\
       letter_box_movie\last.csv", index=False)
```

Listing 6: Extraction of the reviews

This method of defining separate functions for each column allows for tailored extraction procedures. Since the tags and URLs vary for each type of information, we needed to ensure that the correct HTML elements were targeted for each extraction task.
You can find each function in the Jupyter notebook `Extraction.ipynb`.

3 Data Extraction from Rotten Tomatoes

To enhance our movie dataset, we implemented also a web scraping process to extract relevant information about movies from Rotten Tomatoes. This includes details such as critics and audience scores, movie sentiments, and additional metadata like release date, duration, and genres. We applied this method to several URLs, such as https://www.rottentomatoes.com/browse/movies_in_theaters/sort:critic_lowest, resulting in the extraction of data for 460 movies.

3.1 Scraping Rotten Tomatoes Data with Selenium

We began by using Selenium to scrape the Rotten Tomatoes pages dynamically, allowing us to load all available movie data by clicking the "Load More" button until all content was loaded. Here's an overview of the scraping function:

```

def scrape_and_save_to_csv(driver_path, url, output_file):
    # Set up the ChromeDriver service
    service = Service(driver_path)
    options = Options()
    options.add_argument("--start-maximized")
    options.add_argument("--disable-notifications")

    driver = webdriver.Chrome(service=service, options=options)
    driver.get(url)
```

```python
    # Load all data by clicking "Load More"
    while True:
        try:
            load_more_button = WebDriverWait(driver, 10).until(
                EC.element_to_be_clickable((By.CSS_SELECTOR, "button[
                    data-qa='dlp-load-more-button']"))
            )
            load_more_button.click()
            time.sleep(2)  # Allow time for data to load
        except Exception as e:
            print("No 'Load more' button found or end of list.")
            break

    # Extract the HTML and parse it with BeautifulSoup
    html_content = driver.page_source
    driver.quit()
    soup = BeautifulSoup(html_content, "html.parser")
    movies = soup.find_all("a", {"data-qa": "discovery-media-list-item-
        caption"})

    # Extract movie data
    scraped_data = []
    for movie in movies:
        try:
            movie_name = movie.find("span", {"data-qa": "discovery-media
                -list-item-title"}).text.strip()
            critics_score = movie.find("rt-text", {"slot": "criticsScore
                "}).text.strip()
            audience_score = movie.find("rt-text", {"slot": "
                audienceScore"}).text.strip()
            critics_sentiment = movie.find("score-icon-critics")["
                sentiment"]
            audience_sentiment = movie.find("score-icon-audience")["
                sentiment"]
            scraped_data.append({
                "Movie Name": movie_name,
                "Critics Score": critics_score,
                "Critics Sentiment": critics_sentiment,
                "Audience Score": audience_score,
                "Audience Sentiment": audience_sentiment,
            })
        except Exception as e:
            print(f"Error extracting movie: {e}")

    # Save data to CSV
    with open(output_file, mode="w", newline="", encoding="utf-8") as
        file:
        writer = csv.DictWriter(file, fieldnames=["Movie Name", "Critics
            Score", "Critics Sentiment", "Audience Score", "Audience
           Sentiment"])
        writer.writeheader()
        writer.writerows(scraped_data)

    print(f"Data successfully saved to {output_file}")
```

Listing 7: Extraction from Rotten tomato

The function above handles the process of loading the page, clicking the "Load More" button until all data is visible, extracting the movie names, scores, and sentiments, and saving the data to a CSV file.

3.2 Fetching Detailed Movie Information

Next, we fetched additional detailed information for each movie from Rotten Tomatoes using a direct URL to each movie page. This information includes the synopsis, release date, duration, genres, and critics reviews. We used the following function to retrieve this data:

```python
def get_movie_info(movie_name):
    # Clean movie name for URL formatting
    movie_url_name = movie_name.replace(" ", "_").lower()
    url = f"https://www.rottentomatoes.com/m/{movie_url_name}"

    # Fetch the page content
    response = requests.get(url)
    soup = BeautifulSoup(response.text, "html.parser")

    # Extract the synopsis
    synopsis = soup.find("drawer-more", {"maxlines": "2"})
    synopsis_text = synopsis.find("rt-text", {"slot": "content"}).text.
        strip() if synopsis else "N/A"

    # Extract metadata (release date, duration, genres)
    metadata = soup.find_all("rt-text", {"slot": "metadataProp"})
    release_date = metadata[0].text.strip() if metadata else "N/A"
    duration = metadata[1].text.strip() if len(metadata) > 1 else "N/A"
    genres = [genre.text.strip() for genre in soup.find_all("rt-text", {
        "slot": "metadataGenre"})]

    # Extract reviews
    reviews = []
    review_section = soup.find("section", {"aria-labelledby": "critics-
        reviews-label"})
    if review_section:
        review_cards = review_section.find_all("media-review-card-critic
            ")
        for card in review_cards:
            critic_name = card.find("rt-link", {"slot": "displayName"}).
                text.strip()
            review_text = card.find("rt-text", {"slot": "content"}).text
                .strip()
            reviews.append((critic_name, review_text))

    return {
        "movie_name": movie_name,
        "synopsis": synopsis_text,
        "release_date": release_date,
        "duration": duration,
        "genres": genres,
        "reviews": reviews
    }

movies_info = df_combined["Movie Name"].apply(get_movie_info)
df_movie_details = pd.json_normalize(movies_info)
```

Listing 8: Details from rotten tomato

The above code defines a function that constructs the movie URL, sends a request to retrieve the page, and extracts various details, including the synopsis, release date, genre, and critics reviews.

4 Cleaning our Data

Data cleaning is an essential step in any data analysis process to ensure the accuracy, consistency, and usability of the data. In this case, cleaning the Rotten Tomatoes and Letterboxd datasets was necessary to standardize the format of the data, making it easier to compare and analyze. This involved transforming the critics' and audience scores from percentages into ratings out of 5, which ensures that the data is on the same scale. Additionally, the column names across both datasets were standardized to match the same format and naming convention.

4.1 Cleaning Rotten Tomato dataset

After extracting the data from Rotten Tomatoes, the next step is to clean the dataset before merging it with other data. The Rotten Tomatoes dataset initially includes the critics' scores and audience scores as percentages. To make the data more usable and comparable across different platforms, these percentages were transformed into ratings out of 5.
To achieve this transformation, I first removed the percentage sign and converted the values into integers. Then, the percentages were divided by 100 to convert them into decimal values. Finally, I calculated the average score by combining both the critics' score and the audience score, and then scaling it to a rating out of 5.
The following code was used to clean the Rotten Tomatoes data:

```
merged_df['CRITICS SCORE'] = merged_df['CRITICS SCORE'].str.replace('%',
     '').astype(int)
merged_df['AUDIENCE SCORE'] = merged_df['AUDIENCE SCORE'].str.replace('%
    ', '').astype(int)

# Convert percentages to decimal
merged_df['CRITICS SCORE'] = merged_df['CRITICS SCORE'] / 100
merged_df['AUDIENCE SCORE'] = merged_df['AUDIENCE SCORE'] / 100

# Calculate the rating out of 5
merged_df['RATING_SUR_5'] = ((merged_df['CRITICS SCORE'] + merged_df['
    AUDIENCE SCORE']) / 2) * 5
```

Listing 9: Cleaning the Rating Column of rotten tomato data

4.2 Cleaning Letterboxd Data

In the process of cleaning the Letterboxd data, several important transformations were applied to ensure consistency and to prepare the data for analysis. Specifically, columns like "RATING" and "DURATION" were initially in text format, requiring conversion to

numeric values for further analysis.
To clean the "DURATION" column, the following steps were performed:

- The "DURATION" column was first converted to a string format using

  $$df["DURATION"].astype(str)`$$

  to ensure that it could be processed correctly.

- We then extracted the numeric values representing the duration (in minutes) from the text data. If the duration was in the form "Xh Ym", such as "1h 45", we applied a transformation to extract the number of hours and minutes, then convert the total time into minutes. The code below was used for this transformation:

```
    df["DURATION_MIN"] = df["DURATION"]
    .str.extract(r"(\d+)h\s*(\d+)min", expand=False)   # Capture
       hours and minutes
    .apply(lambda x: int(x[0]) * 60 + int(x[1]) if pd.notna(x[0])
       and pd.notna(x[1]) else None, axis=1)
```

  This formula multiplies the number of hours by 60 and adds the minutes to calculate the total duration in minutes.

- For cases where only hours or minutes were provided (e.g., "1h" or "45min"), similar transformations were applied to ensure that the duration was represented correctly in minutes.

- Similarly, the "RATING" column was transformed into a numeric rating on a 5-point scale by extracting the numerical values from the string format and converting them to floats with this code:

```
    df["RATING_SUR_5"] = df["RATING"].str.extract(r"(\d+(\.\d+)?)",
        expand=False)[0].apply(lambda x: float(x) if pd.notna(x)
       else None)
```

- Finally, the original "DURATION" and "RATING" columns were dropped from the dataset since they were no longer necessary for further analysis.

Additionally, other columns such as "GENRE", "PRODUCERS", and "CAST" were standardized. This was done to ensure consistency in the way data is represented across both Rotten Tomatoes and Letterboxd, facilitating easier comparison and merging.
By cleaning and standardizing these data elements, we can now combine information from both sources, ensuring that the final dataset is consistent and ready for further analysis.

5 Data Analysis

In the data analysis phase of this project, we explored various aspects of the dataset to gain insights and identify key trends. Below are the major analyses and visualizations performed:

5.1 Distribution of Movie Genres

We examined the distribution of movie genres to identify the most frequently occurring genre. This analysis helped us understand the dataset's composition and highlight which genres dominate the dataset.

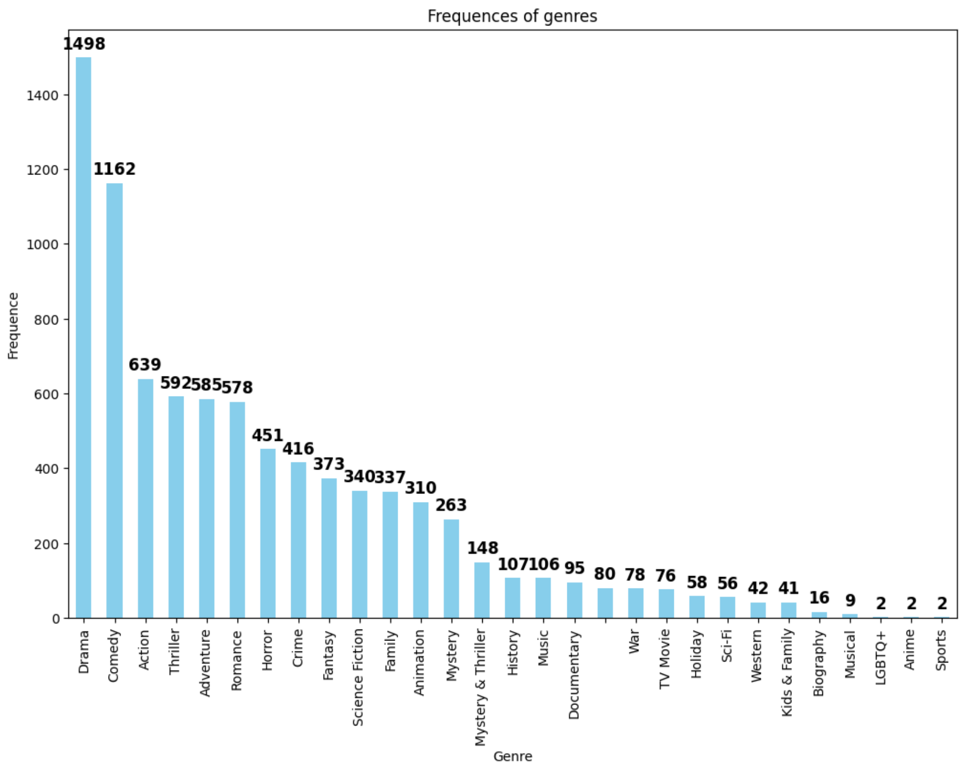


Figure 1: Distribution of Movie Genres

The distribution of movie genres in the dataset reveals several key insights:

- **Dominance of Popular Genres:** Drama is the most prevalent genre with 1,498 movies, followed by Comedy (1,162) and Action (639). This indicates that these genres dominate the dataset, likely reflecting audience demand and market trends. Thriller (592) and Adventure (585) also have a strong presence, showcasing their popularity in the entertainment industry.
- **Niche Genres:** Genres like Western (42), Biography (16), Musical (9), and LGBTQ+ (2) are underrepresented. This suggests that either fewer movies are made in these categories, or they are less frequently included in the dataset.

The genre distribution provides essential insights that can significantly enhance the modeling phase:

- **Incorporating Genre as a Feature:** Each movie's genre(s) should be included as a feature in the model. Multi-label encoding **one-hot encoding** can effectively represent movies with multiple genres.
- **Exploring Genre-Specific Trends:** Certain genres may consistently perform better (e.g., higher ratings for Drama or lower ratings for Horror). Adding weights or interaction terms for genres can refine predictions.

5.2 Average Movie Duration

The average duration of movies was calculated to provide a general idea of how long movies in the dataset typically are. This helped to identify any anomalies or trends related to movie lengths.

Category	**Average Duration (minutes)**
All Movies	107.02
Well-rated Movies	108.50
Poorly-rated Movies	101.67

Table 1: Average durations of movies by category.

5.3 Most Repeated Actors in Top-Rated Movies

We identified the actors who frequently appeared in the most highly rated movies in the dataset. The following list shows the top 10 actors along with the number of movies they appeared in:

- Tom Cruise - 31 movies
- Samuel L. Jackson - 24 movies
- Mickie McGowan - 23 movies
- Ving Rhames - 22 movies
- Brad Pitt - 21 movies
- Jr. - 21 movies
- Sherry Lynn - 21 movies
- Robert De Niro - 20 movies
- Ralph Fiennes - 19 movies
- Jack Angel - 19 movies
- Simon Pegg - 19 movies
- Alec Baldwin - 18 movies
- Bess Flowers - 18 movies
- Bob Bergen - 18 movies

These actors are the ones who most frequently appeared in the top-rated films, reflecting their prominence in well-received movies.
For the modeling phase, the presence of certain actors can significantly impact the success of a movie. Based on our analysis, we hypothesize that movies featuring well-known and highly-rated actors are more likely to receive positive ratings. To incorporate this factor into the model, we plan to add these actors as an additional feature in the dataset. By

including them as columns in the model, we will test the hypothesis that movies with certain actors tend to be rated higher, potentially indicating their contribution to the movie's overall success. If an actor from the list above appears in a movie, it is likely that the film will have a better rating, as these actors are commonly associated with high-quality films.

5.4 Distribution of Movie Ratings

The dataset's movie ratings were classified into three categories based on the following criteria:

- **Positive**: Rating ≥ 3.5
- **Neutral**: $3 \leq$ Rating < 3.5
- **Negative**: Rating < 3

A bar chart was created to display the number of movies in each of these categories. This helped visualize the distribution of movie ratings across the dataset.

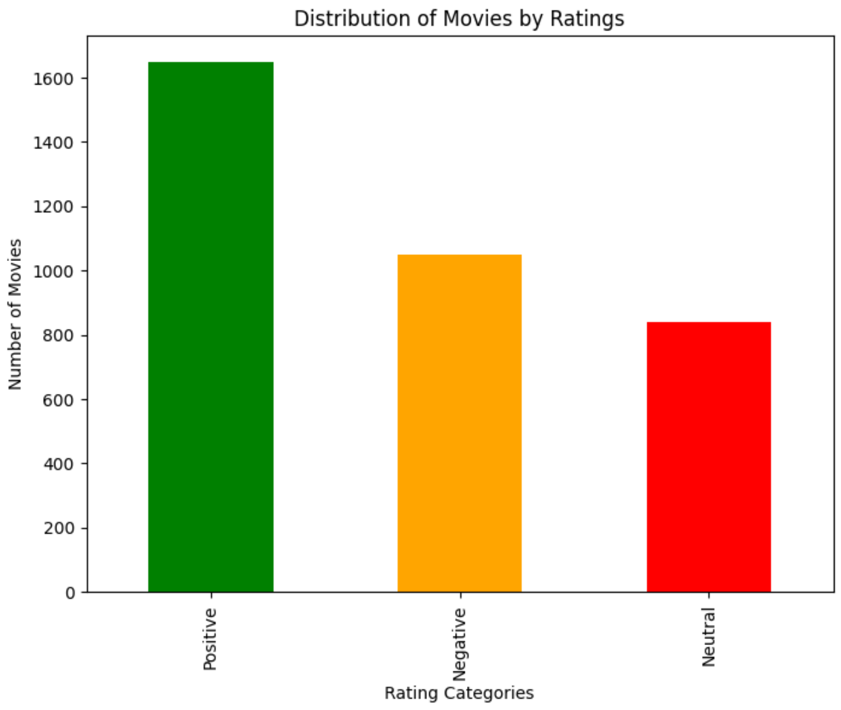


Figure 2: Distribution of Movie Rating

The following numbers shows the distribution of movies in each category:

- **Positive**: 1648 movies
- **Neutral**: 838 movies
- **Negative**: 1049 movies

The distribution of movies by rating category provides valuable insights for the modeling phase of this project. With the majority of movies classified as Positive (1648 movies), followed by Negative (1049 movies) and Neutral (838 movies), this suggests that most movies tend to receive favorable ratings. For the modeling part, we aim to predict the rating of each movie based on a variety of features.

5.5 Sentiment Distribution in all the Reviews

In this analysis, we categorized the movie reviews into three sentiment categories: Positive, Neutral, and Negative. The sentiment analysis was conducted using **the nltk library and the VADER sentiment analysis tool**, which calculates a compound score for each review. Based on the compound score, reviews were classified into one of the three categories.

```python
def classify_reviews_nltk(reviews):
    sentiments = []
    for review in reviews:
        # Calculate the score
        scores = sia.polarity_scores(review.strip())
        if scores['compound'] > 0.1:
            sentiments.append('positive')   #positif if compound > 0.1
        elif scores['compound'] >= 0 and scores['compound'] <= 0.1:
            sentiments.append('neutral')    #neutre if compound entre 0
                et 0.1
        else:
            sentiments.append('negative')   #negatif if compound < 0

    return sentiments
```

Listing 10: Function to extract the sentiments

The total number of reviews processed was 54,770, with the following distribution across sentiment categories:

- **Positive reviews**: 25,389
- **Neutral reviews**: 15,129
- **Negative reviews**: 14,234

A new column, `MAJORITY_SENTIMENT`, was created to capture the dominant sentiment for each movie based on the majority sentiment of its reviews. This allows us to understand the overall reception of each movie in terms of audience sentiment. The sentiment analysis indicates that the majority of reviews are **positive**, with 25,389 positive reviews. However, there is still a significant portion of **neutral** (15,129) and **negative** (14,234) reviews, suggesting that some movies may have mixed or unfavorable reception from their audience. For the modeling phase, the `MAJORITY_SENTIMENT` column will serve as an important feature. By considering sentiment distribution, we can enhance the model's ability to predict movie ratings more accurately. Movies with a higher number of positive reviews will likely receive higher ratings, while those with more neutral or negative reviews may reflect lower ratings. This feature, along with other attributes such as genre, actors, and synopsis, will help to refine the model's predictions and understand audience preferences.

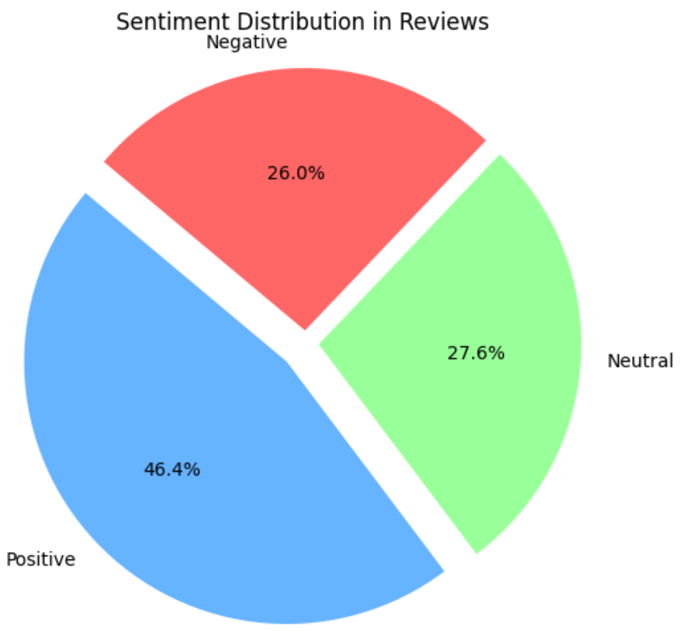


Figure 3: Distribution of reviews

5.6 Sentiment Distribution for Each Movie

To understand the sentiment trends for individual movies, we analyzed the most frequent sentiment associated with each movie. A bar plot was used to visualize the distribution of these sentiments across all movies.

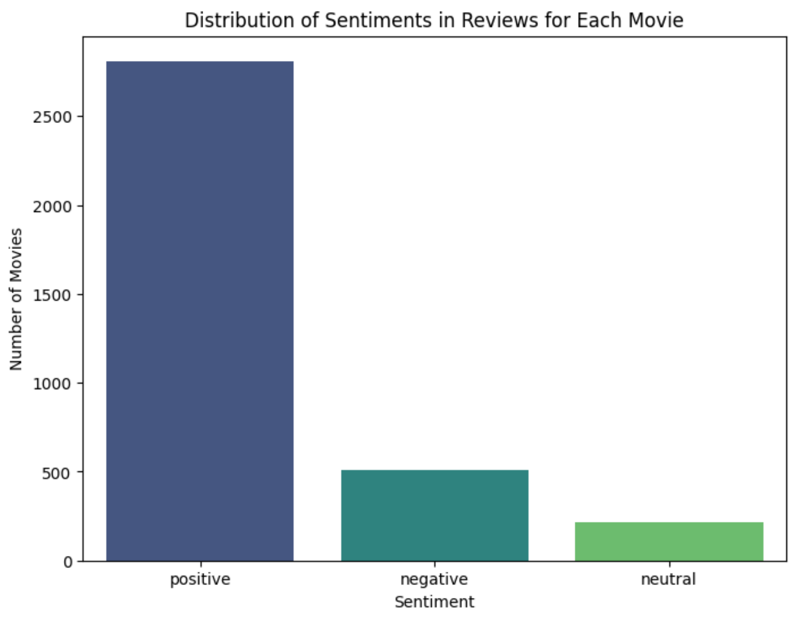


Figure 4: Distribution of the most dominant sentiment in each movie

To gain deeper insights into sentiment trends for individual movies, we analyzed the most frequent sentiment associated with each film. A bar chart was created to visualize the distribution of these dominant sentiments across all movies. A new column, `MOST_FREQUENT_SENTIMENT`, was added to capture the most commonly occurring sentiment for each movie based on the MAJORITY_SENTIMENT column, which aggregates the sentiments from all reviews of a given movie. This approach highlights the overall sentiment trend for each film, offering valuable context for further analysis.

5.7 Emotion Analysis

Emotion analysis is a critical component of our project. It aims to identify the dominant emotion in each review using an emotion classification model.
To perform this analysis, we used the function analyze_emotions, defined as follows:

```
emotion_analyzer = pipeline("text-classification", model="bhadresh-
    savani/bert-base-uncased-emotion", return_all_scores=True)

def analyze_emotions(reviews):
    emotions = []
    for review in tqdm(reviews, desc="Analyzing emotions", unit="review"
        ):
        result = emotion_analyzer(review.strip())  # Returns emotion
            predictions
        if isinstance(result, list) and len(result) > 0 and isinstance(
            result[0], dict):
            emotions.append(result[0]['label'])  # Accessing 'label'
        else:
            emotions.append(None)  # Handling unexpected format
    return emotions
```

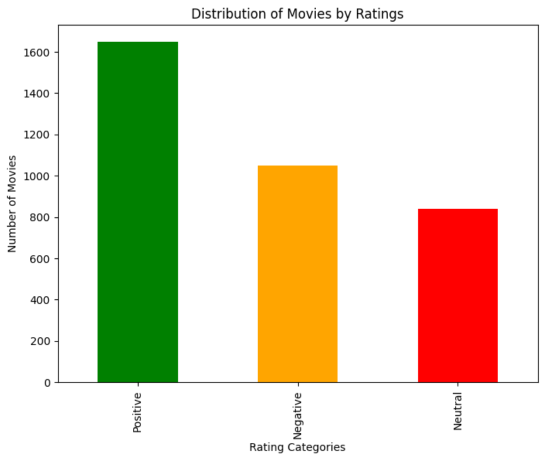
This function iterates through each review in the REVIEWS column and uses an emotion analysis model to identify the primary emotion associated with each review. The progress bar tqdm is used to display the processing status. The model returns a set of scores for each possible emotion, along with a label indicating the dominant emotion. For example, for a given review, the model may produce the following results:

- **anger**: 0.328
- **disgust**: 0.283
- **fear**: 0.087
- **joy**: 0.006
- **neutral**: 0.202
- **sadness**: 0.070
- **surprise**: 0.022

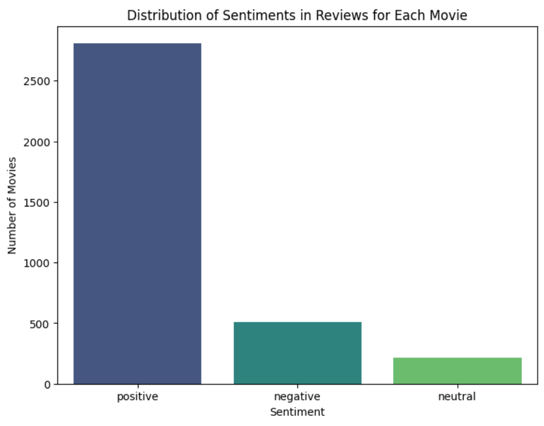
In this example, the dominant emotion is **anger**, as it has the highest score.
To visualize the results, we added a column EMOTIONS to the DataFrame, which contains the dominant emotion for each review.

6 Comparison of Sentiment Evaluation Methods

The results from the two different methods of evaluating movie sentiments—based on ratings versus review sentiment distribution—offer interesting insights into how movie quality is perceived and categorized.



(a) Distribution of movie rating



(b) Distribution of the most dominant sentiment in each movie

Figure 5: Comparison of two graphics.

Key Observations :

- **Disparity in Positive Movies**: The review-based method identifies significantly more movies (more than 2,500) as "Positive" compared to the rating-based method (1,648). This suggests that audiences may use more nuanced expressions in reviews that do not align perfectly with numerical ratings.
- **Fewer Neutral and Negative Movies in Reviews**: The review-based method shows a much lower count of neutral (300) and negative (500) movies. This indicates that textual reviews tend to express stronger opinions (either positive or negative), while ratings might reflect a more moderate or balanced sentiment.
- **Subjectivity in Sentiments**: Reviews provide a qualitative perspective, emphasizing the subjective experiences of viewers. Ratings, on the other hand, offer an aggregated quantitative assessment.
- **Tendency to Give Ratings Without Explanation**: Many users tend to provide numerical ratings but avoid explaining their reasoning in textual reviews. This might lead to a disconnect between ratings and reviews, making it challenging to fully understand the audience's sentiment based on one method alone.
- **Complementary Insights for Modeling**: Combining both methods could improve prediction models. Ratings provide a general sentiment trend, while review analysis adds context and granularity, particularly for movies with mixed or polarizing feedback.

7 Modeling

7.1 Text Preprocessing

For the textual columns such as `SYNOPSIS` and `REVIEWS`, the following preprocessing steps were applied:

- Replaced numbers (e.g., ages, years, quantities) with placeholders such as `AGE`, `YEAR`, and `QUANTITE`.
- Tokenized the text, converted it to lowercase, and removed non-alphanumeric characters.
- Removed stopwords and applied lemmatization to reduce words to their base form.

```
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer
from nltk.tokenize import word_tokenize
stop_words = set(stopwords.words('english'))
lemmatizer = WordNetLemmatizer()

def preprocess_text(text):
    if isinstance(text, str):
        text = re.sub(r'\b\d{1,2} ans\b', 'AGE', text)
        text = re.sub(r'\b\d{4}\b', 'YEAR', text)
        text = re.sub(r'\b\d+\s?(kg|g|lbs|hours|minutes)\b', 'QUANTITE',
            text)
        tokens = word_tokenize(text.lower())
        tokens = [lemmatizer.lemmatize(word) for word in tokens if word.
           isalnum()]
        tokens = [word for word in tokens if word not in stop_words]
        return " ".join(tokens)
    return ""
```

Listing 11: Text Preprocessing Function

7.2 Feature Engineering

Encoding Genres and Cast:

- The `GENRES` and `CAST` columns were split into individual elements.
- Irrelevant entries (e.g., `Show All...` or `Unavailable`) were removed.
- Binary columns were created for the most frequent genres and actors.

```
# Extract unique genres
all_genres = set([genre for sublist in df_copy['GENRES'] for genre in
    sublist])

# Create binary columns for each genre
for genre in all_genres:
    df_copy[f'genre_{genre}'] = df_copy['GENRES'].apply(lambda x: 1 if
        genre in x else 0)

# Filter and encode top 10 actors
df_copy['CAST'] = df_copy['CAST'].apply(lambda x: x.split(', ') if
    isinstance(x, str) else [])
all_casts = [actor for sublist in good_movies['CAST'] for actor in
    sublist]
popular_casts = [actor for actor, _ in Counter(all_casts).most_common
    (10)]
```

```
for actor in popular_casts:
    df_copy[f'ACTEUR_{actor}'] = df_copy['CAST'].apply(lambda x: 1 if
        actor in x else 0)
```

Listing 12: Binary Encoding for Genres and Actors

7.3 Data Preprocessing

The numerical column `DURATION_MIN` is converted into a numeric type, with non-numeric values replaced by `NaN`. Missing values are filled with the mean of the column. We then scale the numerical features using `StandardScaler`.

```
# Convert 'DURATION_MIN' to numeric, replacing non-numeric values with
    NaN
data['DURATION_MIN'] = pd.to_numeric(data['DURATION_MIN'], errors='
    coerce')
data['DURATION_MIN'] = data['DURATION_MIN'].fillna(data['DURATION_MIN'].
    mean())

# Extract numerical columns and scale them
numerical_columns = ['DURATION_MIN']
numerical_data = data[numerical_columns].values
scaler = StandardScaler()
numerical_features = scaler.fit_transform(numerical_data)
```

Listing 13: Data Preprocessing

The binary columns, such as genres, actors, and producers, are already encoded. We extract these features into a `binary_data` array.

```
binary_columns = [col for col in data.columns if col.startswith('genre_'
    ) or col.startswith('ACTEUR_') or col.startswith('PRODUCER_')]
binary_data = data[binary_columns].values
```

Listing 14: Binary Data Extraction

For categorical columns, such as `MOST_FREQUENT_SENTIMENT` and `Rating_Category`, we apply `OneHotEncoder` to perform encoding.

```
from sklearn.preprocessing import OneHotEncoder

categorical_columns = ['MOST_FREQUENT_SENTIMENT', 'Rating_Category']
encoder = OneHotEncoder()
categorical_data = encoder.fit_transform(data[categorical_columns])
```

Listing 15: Categorical Data Encoding

7.4 Model Selection and Evaluation

We combine all features into a single array `X`, and the target variable `y` is the column `RATING_SUR_5`. The data is split into training and testing sets (85%-15% split).

```
from sklearn.model_selection import train_test_split

X = np.hstack([textual_features, numerical_features, binary_data,
    categorical_data.toarray()])
```

```
y = data['RATING_SUR_5']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size
    =0.15, random_state=42)
```

Listing 16: Feature Combination and Train-Test Split

We evaluate several regression models, including `LinearRegression`, RandomForestRegressor, `SVR`, and `HistGradientBoostingRegressor`. Cross-validation is used to assess the performance of each model using negative mean absolute error as the evaluation metric.

```
models = {
    'Linear Regression': LinearRegression(),
    'Random Forest Regressor': RandomForestRegressor(),
    'Support Vector Regressor': SVR(),
    'HistGradientBoosting Regressor': HistGradientBoostingRegressor()
}

results = {}
for name, model in models.items():
    cv_scores = cross_val_score(model, X_train, y_train, cv=5, scoring='
        neg_mean_absolute_error')
    results[name] = -cv_scores.mean()

# Display results
for name, score in results.items():
    print(f"{name}: {score:.4f}")
```

Listing 17: Model Evaluation with Cross-Validation

After comparing the models, we select the best one based on the lowest mean absolute error. The selected model is trained on the full training set, and predictions are made on the test set. We then compute the evaluation metrics: Mean Absolute Error (MAE), Mean Squared Error (MSE), and Root Mean Squared Error (RMSE).

```
best_model = models['HistGradientBoosting Regressor']  # Assuming the
    best model is HistGradientBoosting
best_model.fit(X_train, y_train)

# Predict on the test set
y_pred = best_model.predict(X_test)

# Metrics calculation
mae = mean_absolute_error(y_test, y_pred)
mse = mean_squared_error(y_test, y_pred)
rmse = np.sqrt(mse)
```

Listing 18: Model Training and Metrics Calculation

7.5 Hyperparameter Tuning

After identifying the best model, hyperparameter tuning for the HistGradientBoostingRegressor was performed using grid search. The following key hyperparameters were optimized: `max_iter`, `learning_rate`, and `max_depth`. This optimization process helped to fine-tune the model, ensuring improved performance and better generalization on the test data.

```
from sklearn.model_selection import GridSearchCV
param_grid = {
    'max_iter': [100, 200, 300],
    'learning_rate': [0.01, 0.1, 0.2],
    'max_depth': [3, 5, 7]
}
grid_search = GridSearchCV(HistGradientBoostingRegressor(), param_grid,
   cv=5, scoring='neg_mean_absolute_error')
grid_search.fit(X_train, y_train)
```

Listing 19: Grid Search for Hyperparameter Tuning

7.6 Final Model Training and Evaluation

We train the `HistGradientBoostingRegressor` using the best hyperparameters obtained from the grid search and evaluate its performance on the test set.

```
model = HistGradientBoostingRegressor(
    learning_rate=0.1,
    max_depth=7,
    max_iter=100
)

model.fit(X_train, y_train)

# Predict on the test set
y_pred = model.predict(X_test)

# Metrics calculation
mae = mean_absolute_error(y_test, y_pred)
mse = mean_squared_error(y_test, y_pred)
rmse = np.sqrt(mse)
r2 = r2_score(y_test, y_pred)
```

Listing 20: Final Model Training and Evaluation

7.7 Model Evaluation Metrics

The metrics for the final model after training and hyperparameter optimization are as follows:

Metric	**Value**
Mean Absolute Error (MAE)	0.21
Mean Squared Error (MSE)	0.09
Root Mean Squared Error (RMSE)	0.31
R^2 Score	0.8484

Table 2: Regression Metrics of the Final Model

Analysis of the Results

- **Mean Absolute Error (MAE)**: A MAE of 0.21 indicates that, on average, the model's predictions are off by around 0.21 units. This is considered a good performance for many practical applications, especially when compared to larger deviations.

- **Mean Squared Error (MSE)**: The MSE value of 0.0994 shows that the model's errors, on average, are relatively small. However, MSE is more sensitive to large errors, so a lower MSE suggests that the model is generally accurate.
- **Root Mean Squared Error (RMSE)**: With an RMSE of 0.3154, this model's prediction errors are reasonably small, and the square root helps bring the metric back to the same scale as the target variable. This means the model's error is relatively low.
- **R^2 Score**: The R^2 score of 0.8484 means that approximately 85% of the variation in the target variable can be explained by the model. This is a strong result, indicating that the model fits the data well and can generalize well to new data.

Overall, the model performs with good predictive accuracy, minimal error, and a high level of explained variance. This performance makes it suitable for predicting the target variable in future scenarios.

8 Streamlit App

8.1 Description

This Streamlit app is designed to provide a user-friendly interface for interacting with a machine learning model that predicts movie ratings based on various textual, numerical, and categorical data features. The app leverages a trained machine learning model to predict a movie's rating based on user input. Here's a brief overview of the key features:

- **Textual Data Input**: Users can provide textual information such as the movie's synopsis, reviews, genres, cast, and producers. The app will process this input and extract useful features using TF-IDF vectorization.
- **Numerical Data Input**: The app also accepts numerical inputs, such as the movie's duration, which are normalized for better model performance.
- **Model Prediction**: The model is then used to predict the movie's rating on a scale of 1 to 5, based on the input data.

8.2 How to run the app

This guide will walk you through the steps to run our Streamlit app locally :

- Install the required libraries **pip install -r requirements.txt**
- Execute this command on your terminal : **streamlit run streamlit.py**

8.3 Interface

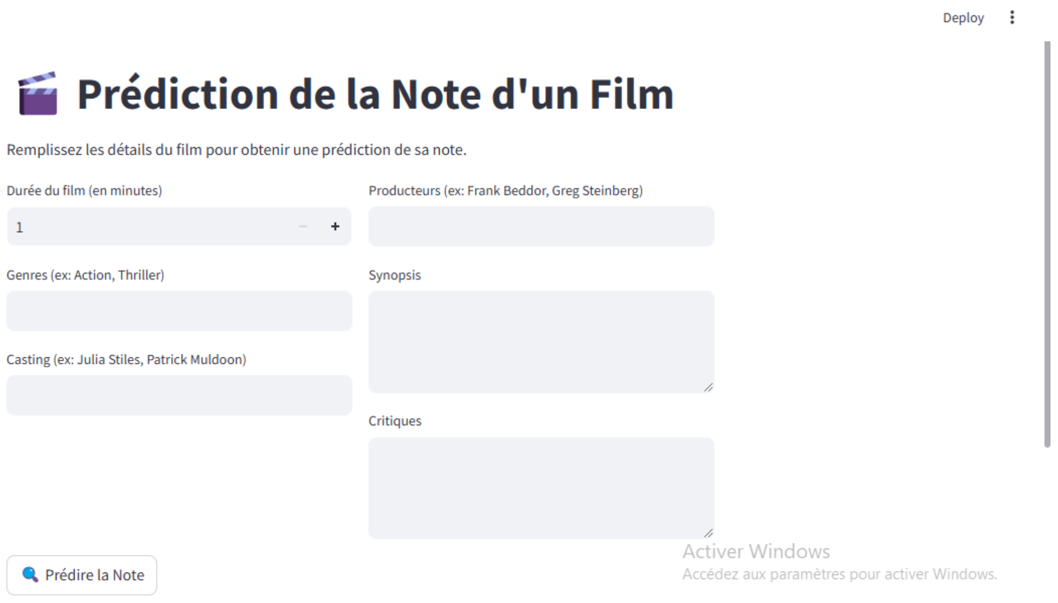
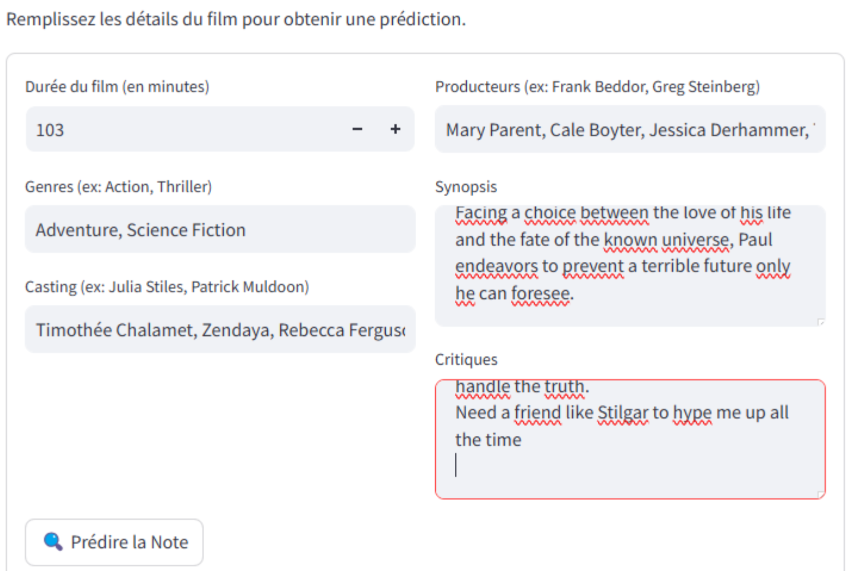

Deploy
🎬 Prédiction de la Note d'un Film
Remplissez les détails du film pour obtenir une prédiction de sa note.
Durée du film (en minutes)
1
Producteurs (ex: Frank Beddor, Greg Steinberg)
Genres (ex: Action, Thriller)
Synopsis
Casting (ex: Julia Stiles, Patrick Muldoon)
Critiques
🔍 Prédire la Note
Activer Windows
Accédez aux paramètres pour activer Windows.

Figure 6: Interface of the streamlit app


Remplissez les détails du film pour obtenir une prédiction.

Durée du film (en minutes): 103

Producteurs (ex: Frank Beddor, Greg Steinberg): Mary Parent, Cale Boyter, Jessica Derhammer,

Genres (ex: Action, Thriller): Adventure, Science Fiction

Synopsis: Facing a choice between the love of his life and the fate of the known universe, Paul endeavors to prevent a terrible future only he can foresee.

Casting (ex: Julia Stiles, Patrick Muldoon): Timothée Chalamet, Zendaya, Rebecca Fergus

Critiques: handle the truth.
Need a friend like Stilgar to hype me up all the time

Figure 7: Interface of the streamlit app

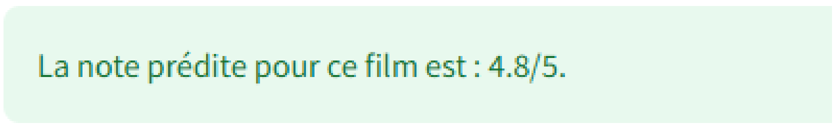

La note prédite pour ce film est : 4.8/5.

Figure 8: Result

9 Conclusion

In this project, we aimed to build a regression model to predict a target variable based on a combination of textual, numerical, binary, and categorical features. After performing

data preprocessing, feature engineering, and model training, we explored several machine learning algorithms to identify the most effective model for our task.
The `HistGradientBoostingRegressor` emerged as the best model, outperforming alternatives such as Linear Regression, Random Forest, and Support Vector Regressor. By fine-tuning the model's hyperparameters using GridSearchCV, we achieved strong performance on the test set.
The system utilized natural language processing (NLP) techniques for sentiment analysis and machine learning algorithms to predict the overall rating of a movie from critic reviews. By analyzing various features such as the sentiment of the reviews, movie genres, and historical ratings, the model was able to provide valuable insights into how critics assess films.
The results demonstrated that sentiment analysis, combined with relevant metadata, effectively predicted movie ratings and revealed trends in critic reviews. Additionally, this project enhanced my skills in data preprocessing, feature extraction, and model evaluation. Future work could focus on further refining the model by integrating more advanced NLP techniques or exploring deeper collaborative filtering methods to improve the accuracy of movie rating predictions.