Web Scraping and Movie Analytics

Type of CW	Take Home Assignments (THA)
Deadline	@April 25, 2024
Lecturer	ProgrammerRDAI
Week	8

Webpage Selection

Introduction

IMDb (Internet Movie Database), founded in 1990 by a British programmer, is a reputed platform that hosts information related to movies, television shows, video games, and reality shows. It includes information about millions of films and television programs, as well as their cast and crew. Currently, IMDb is a subsidiary of Amazon. For web scraping movie data, IMDb is an excellent choice. With its vast user base and constant contributions, it was selected for this project. (Lewis, 2024; Chhikara, 2022; Fisher, 2022)

Data Availability

With over 17.86 million movie titles, 13.14 million human records and 83 million users, IMDb offers one of the biggest movie-related data collections (*How to Scrape Movie Data from IMDb*, no date). With a vast variety of data points, IMDb is an excellent resource for data-driven or analytical projects regarding movies and entertainment.

Relevance

IMDB has real-time updates about recent updates in the movie industry, which is a major benefit of IMDb because it has a constantly updated database. IMDb is a perfect platform for analysing trends in movies and shows (How to Scrape Movie Data from IMDb, no date).

Reliability

IMDb is known for its accuracy and reliability. To ensure reliability, IMDb only allows certified users to interact with the database itself. This ensures that the data scraped from IMDb is both current and trustworthy (Chhikara, 2022).

Ease of Access and Scraping

IMDb’s website has a consistent format and layout, allowing for the specific written scraping scripts to be used throughout for a long time, in turn making it relatively easy to navigate and scrape the websites with the use of web scraping tools.

Conclusion

In conclusion, IMBd has been chosen for this project due to its extensive database, relevance, reliability, and ease of access and scraping, making it an ideal choice for this project.

Web Scraping Implementation

Github Repository : https://github.com/Programmer-RD-AI/MovieDataScraper

Libraries

BeautifulSoup4

BeautifulSoup, created by Leonard Richardson in 2004, was designed to simplify parsing. It has become one of the most popular due to its ease of use and efficiency in navigating through web pages. It specializes in parsing documents, making them useful for extracting data from static web pages (not changing / still). BeautifulSoup4 was used in this project due to its ease of navigation with parse trees and simple methods to manipulate elements, and since the objective was mainly web scraping, interaction with the web application was not required In turn, frameworks such as selenium was discarded from possible libraries to be used. With BeautifulSoup’s efficient and high-speed ability to extract data, its supportive community, and thorough documentation, it was chosen as the web scraping framework for this project. (codedamn news, 2024)

Requests

Request is a library built on top of the `http.client` module (part of Python's standard libraries) and is a widely used, popular, and standard Python library for making HTTP requests. It helps abstract the complexity and provides an easy interface to interact with APIs. Requests were chosen to make requests in this project to get the HTML content of web pages due to its simplicity, support for various methods (such as GET, POST, PUT, DELETE, etc.), documentation, and community support (Python, n.d.; Kim, 2023).

Code Structure and Techniques

Scraped Data

- Cast: movieID, castName, castType, castRole, castUrl
- FAQ: movieID, faqTitle, faqContent, faqUrl
- Movie: movieID, movie description, movie scores, user reviews, critic reviews, metascore, nominationUrl, storyLine, taglines, genres, certificate, cast, faq, videos, photos, technical specifications
- Movies Basic Info: movieID, movieImg, movieMoreUrl, movieTitle, movieYear, movieTime, movieAvgRating, movieRatingCount
- Photos: movieID, photoTitle, photoUrl
- Similar Movies: movieID, movieTitle, movieUrl
- Technical Specifications: movieID, specKey, specValue, specLink
- Videos: movieID, videoTitle, videoSubTitle, videoLink

Data Storage

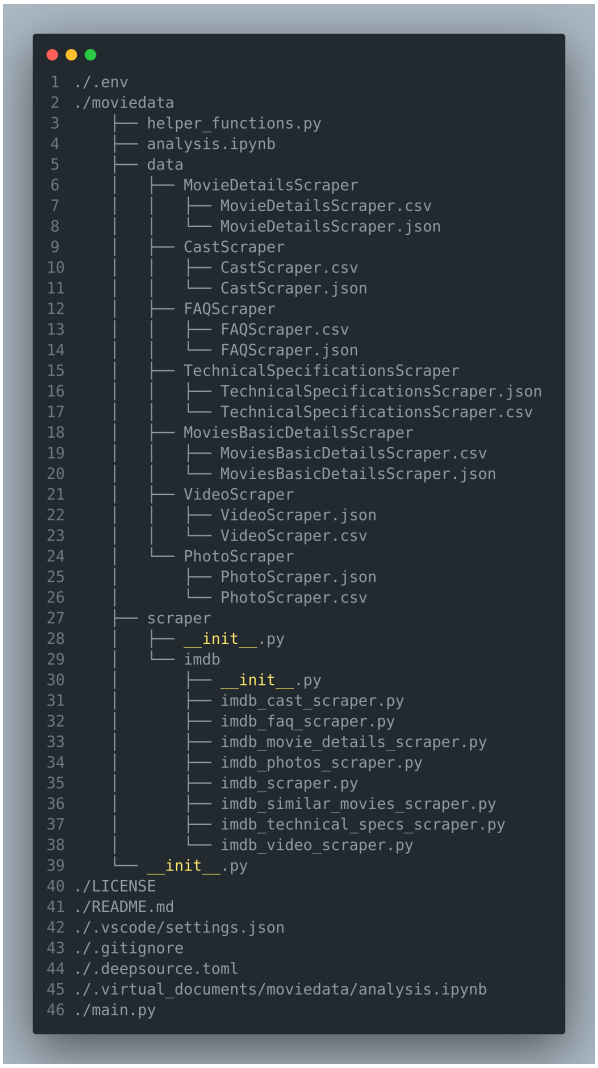
- Strategy: The current data is stored locally with the use of JSON and CSV file formats; both contain the same data but are exported after web scanning to ensure that the data exists

and in case of an error regarding one format, the other will always store the information.

- Schema: The following image describes the schema design of the files; it shows how all of the files are interconnected with one another.
- Future Considerations: As for future implementation, the system would have an advantage by using a cloud service so that the entire analytical process can be automated and constantly updated, allowing for constantly updated analytics regarding the scraped data.

Code Organization

- The codebase follows a well-organized approach while allowing for easy navigation and maintenance.
- `.env` file holds all of the environmental variables
- `data` directory has all of the data divided into subdirectories in both CSV and JSON formats
- `scraper` directory contains the scripts for web scraping IMDb
- The project has been structured in such a manner that it is easily expandable, and it used a variety of reusable modules, especially in `helper_functions.py` and `imdb_scraper.py`



- Overall, the code base structure follows best practices, allowing for a modular, maintainable, and scalable code base. This organized structure allows for future enhancements and feature additions.

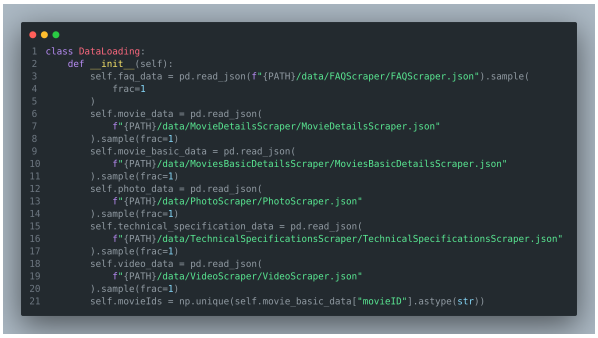
Documentation

The project’s documentation has been structured to ensure ease of understanding and usability. Each Python file is thoroughly documented using docstrings and inline comments that follow the Google Python Style Guide to ensure clarity in the methods' functionality, parameters, and return values. Additionally, a thorough README.md file is created as a general guide to the project with setup instructions and other details.

Data Cleaning and Preprocessing

Loading Pandas DataFrame

- The given code snippet contains a class called: `DataLoading`, which handles all required processes regarding loading data. 6 total data frames are imported, and they are set as class global variables; the files are in the type JSON file when the data is loaded; they are randomly shuffled, so the data



exploration process is much easier; and finally, the unique movieId IDs are taken from a data frame.

Data Cleaning & Justification

- The data that is loaded with the use of `DataLoading` requires cleaning to make sure that it is fit for descriptive analysis. In turn, another class called `DataCleaning`, which inherits from the `DataLoading` class, is used to help streamline the data cleaning process. By using inheritance, the data is loaded and can be used throughout the `DataCleaning` class. The given `self.faq_stats` and `self.price_stats` are used to store the cleaned data.

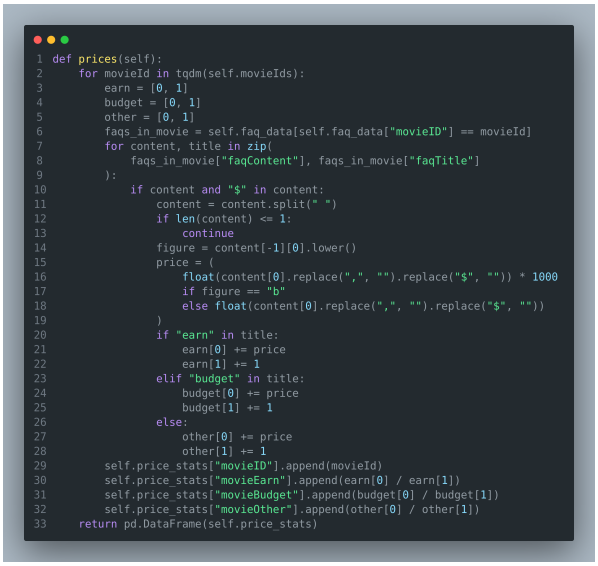
```
class DataCleaning(DataLoading):
    def __init__(self):
        super().__init__()
        self.faq_stats = {"movieID": [], "noOfFAQs": []}
        self.price_stats = {
            "movieID": [],
            "movieEarn": [],
            "movieBudget": [],
            "movieOther": [],
        }
```

- The given function `type_faq_counter` updates the `self.faq_stats` dictionary. By adding the count of the number of FAQs and their specific type of question, the first word is extracted from the given sentence (`faqTitle`) and then a column is made if it doesn't already exist regarding the

```
def type_faq_counter(self):
    for movieId in tqdm(self.movieIds):
        faqs_in_movie = self.faq_data[self.faq_data["movieID"] == movieId]
        self.faq_stats["movieID"].append(movieId)
        self.faq_stats["noOfFAQs"].append(len(faqs_in_movie))
        for faq in faqs_in_movie["faqTitle"]:
            if not faq:
                continue
            name = faq.split(" ")[0].lower() + "Counter"
            if name not in self.faq_stats:
                self.faq_stats[name] = [0] * len(self.movieIds)
            self.faq_stats[name][np.where(self.movieIds == movieId)[0][0]] += 1
    return pd.DataFrame(self.faq_stats)
```

specific type of first word and then the count is updated for each movie.

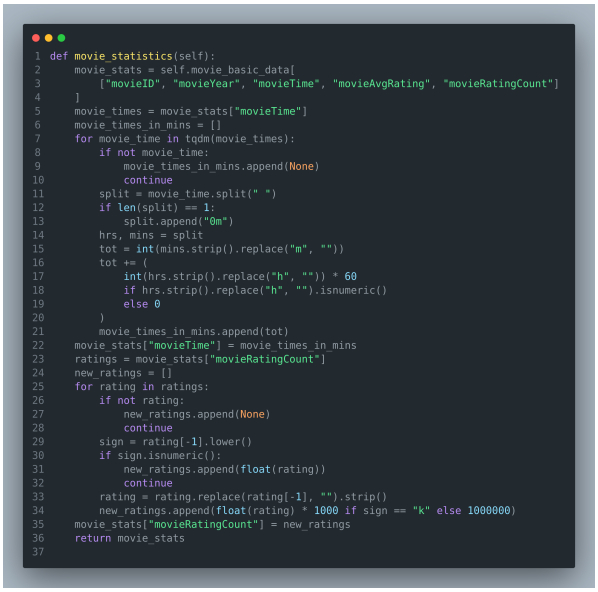
- The `prices` method of the `DataCleaning` class loops through each movie ID in the datasets and extracts the prices mentioned in the FAQs for a specific movie. This is done by iterating over the `faqContent` and `faqTitle` columns of the `faq_data` data frame. If the $ is in the faqContent iterated value, then the method checks if either earn or budget is in the `t`itle. If they are, then the prices are added to the specific list that tracks the price of earn, budget, and other, and if neither `earn` nor `budget` is in the title, then the price is added to the `other` section. Note: All of the prices are in millions.



```
def prices(self):
    for movieId in tqdm(self.movieIds):
        earn = [0, 1]
        budget = [0, 1]
        other = [0, 1]
        faqs_in_movie = self.faq_data[self.faq_data["movieID"] == movieId]
        for content, title in zip(
            faqs_in_movie["faqContent"], faqs_in_movie["faqTitle"]
        ):
            if content and "$" in content:
                content = content.split(" ")
                if len(content) <= 1:
                    continue
                figure = content[-1][0].lower()
                price = (
                    float(content[0].replace(",", "").replace("$", "")) * 1000
                    if figure == "b"
                    else float(content[0].replace(",", "").replace("$", ""))
                )
                if "earn" in title:
                    earn[0] += price
                    earn[1] += 1
                elif "budget" in title:
                    budget[0] += price
                    budget[1] += 1
                else:
                    other[0] += price
                    other[1] += 1
        self.price_stats["movieID"].append(movieId)
        self.price_stats["movieEarn"].append(earn[0] / earn[1])
        self.price_stats["movieBudget"].append(budget[0] / budget[1])
        self.price_stats["movieOther"].append(other[0] / other[1])
    return pd.DataFrame(self.price_stats)
```

- The code is written in a way that is easy to understand and maintain, and with the use of the `tqdm` library, a progress bar is shown to make it easier to track overall progress.

- The `movie_statistics` method is a part of the `DataCleaning` class. This method contains multiple processes to ensure the data is clean, such as converting the time of hours and minutes into minutes only and converting the movie rating count into a float. The following describes how each process works in more details

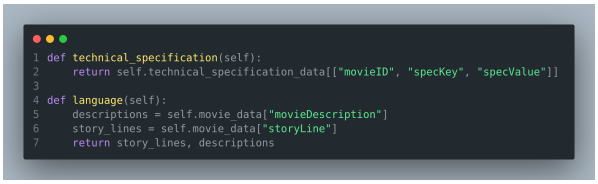


```python
def movie_statistics(self):
    movie_stats = self.movie_basic_data[
        ["movieID", "movieYear", "movieTime", "movieAvgRating", "movieRatingCount"]
    ]
    movie_times = movie_stats["movieTime"]
    movie_times_in_mins = []
    for movie_time in tqdm(movie_times):
        if not movie_time:
            movie_times_in_mins.append(None)
            continue
        split = movie_time.split(" ")
        if len(split) == 1:
            split.append("0m")
        hrs, mins = split
        tot = int(mins.strip().replace("m", ""))
        tot += (
            int(hrs.strip().replace("h", "")) * 60
            if hrs.strip().replace("h", "").isnumeric()
            else 0
        )
        movie_times_in_mins.append(tot)
    movie_stats["movieTime"] = movie_times_in_mins
    ratings = movie_stats["movieRatingCount"]
    new_ratings = []
    for rating in ratings:
        if not rating:
            new_ratings.append(None)
            continue
        sign = rating[-1].lower()
        if sign.isnumeric():
            new_ratings.append(float(rating))
            continue
        rating = rating.replace(rating[-1], "").strip()
        new_ratings.append(float(rating) * 1000 if sign == "k" else 1000000)
    movie_stats["movieRatingCount"] = new_ratings
    return movie_stats

```

- When converting the time from hrs and mins into mins only, a for loop is used to iterate through all of the movie times, and then if the movie_time isn't None, it continues by splitting the movie time by “”, and if the length is 1, an `0m` element is added to ensure the rest of the code works. The split data is then set to 2 variables, hrs and mins, and then the minute variable’s string characters such as m and empty spaces are removed and converted to an integer, and the same with hrs.
- When converting the ratings into floats, the original

ratings contained k, m, or no specification of the amount, with the use of a loop to iterate through ratings and then convert the `k` (meaning thousand) and `m` (millions) into normal numbers and then store them.

- `technical specification` and `language` methods are simple methods that are used as helper functions that are used later on for descriptive statistics



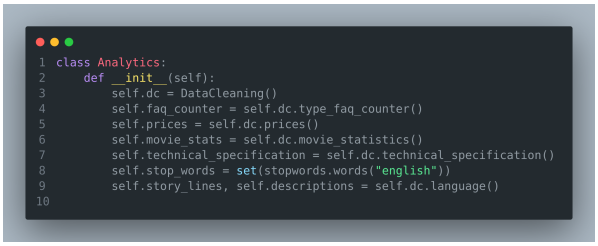
```
def technical_specification(self):
    return self.technical_specification_data[["movieID", "specKey", "specValue"]]

def language(self):
    descriptions = self.movie_data["movieDescription"]
    story_lines = self.movie_data["storyLine"]
    return story_lines, descriptions
```

Descriptive Statistics

Code Walkthrough

`Analytics` Initialization

- The class `Analytics` contains all the analytical methods that have been used in the descriptive statistics class. The a`nalytics` class makes use of matplotlib.pyplot (plt) and seaborn (sns) for analytical purposes. The `DataCleaning` class is used to get the required data to

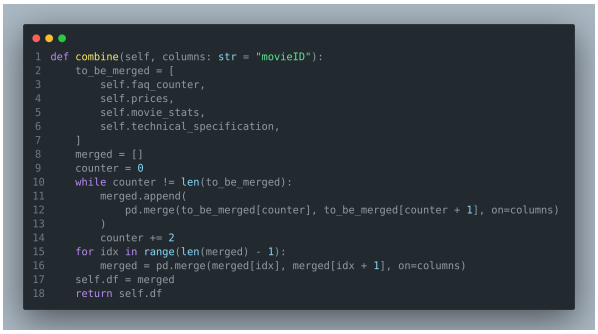


```
class Analytics:
    def __init__(self):
        self.dc = DataCleaning()
        self.faq_counter = self.dc.type_faq_counter()
        self.prices = self.dc.prices()
        self.movie_stats = self.dc.movie_statistics()
        self.technical_specification = self.dc.technical_specification()
        self.stop_words = set(stopwords.words("english"))
        self.story_lines, self.descriptions = self.dc.language()

```

analyze. The given image shows the initialization process of the class; it first creates an instance of the DataCleaning class and then gets the required data for visualization purposes, such as `faq_counter`, `prices`, `technical_specification`, `language`, and `movie_stats`. Additionally, self.stop_words is initialized, which is used with the WorldCloud Library to visualise the most used words in a given set of sentences.

`combine`

- The `combine` method of the `Analytics` combines all of the data frames that taken from the `DataCleaning` instance. It uses `pd.merege` to combine these data frames. The parameter `columns` choose which column is common and then the data is merged according to that. The combined data frame is set to self.df and returned as well.



```python
def combine(self, columns: str = "movieID"):
    to_be_merged = [
        self.faq_counter,
        self.prices,
        self.movie_stats,
        self.technical_specification,
    ]
    merged = []
    counter = 0
    while counter != len(to_be_merged):
        merged.append(
            pd.merge(to_be_merged[counter], to_be_merged[counter + 1], on=columns)
        )
        counter += 2
    for idx in range(len(merged) - 1):
        merged = pd.merge(merged[idx], merged[idx + 1], on=columns)
    self.df = merged
    return self.df
```

correlation_matrix

- The function `correlation_matrix` produces the correlation of all the numerical columns in the data frame. The non-numerical columns movieID, specKey, and specValue are removed with the use of `drop()`, then the correlation is calculated, saved, and finally returned from the function.

```python
def correlation_matrix(self):
    correlation_matrix = self.df.drop(
        ["movieID", "specKey", "specValue"], axis=1
    ).corr()
    correlation_matrix.to_json(f"{PATH}/data/correlation_matrix.json")
    return correlation_matrix
```

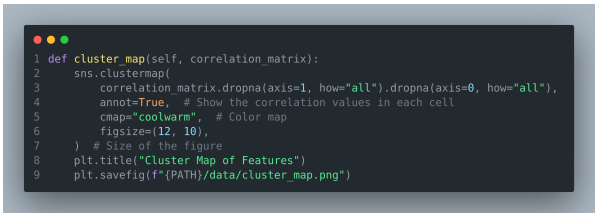
heatmap

- This function is made to generate the correlation heatmap visualization of the correlation matrix of features. First, the style of seaborn is set to “white”, then a plt. figure of size (12,10) is confirmed then the sns.heatmap function is called while passing the correlation_mtrix and other parameters which change the visualizations produced. A title is added to the figure and then the visualization is saved as a.png file.

```python
def heatmap(self, correlation_matrix):
    # Set the style
    sns.set(style="white")
    plt.figure(figsize=(12, 10))
    sns.heatmap(
        correlation_matrix, annot=True, cmap="coolwarm", fmt=".1f", linewidths=0.5
    )
    plt.title("Correlation Heatmap of Features")
    plt.savefig(f"{PATH}/data/heatmap.png")
```

cluster_map

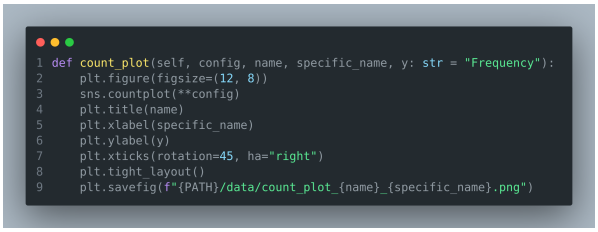
- The cluster_map function produces a cluster map with the use of the clutermap method using seaborn. It removes any rows and columns with missing values. An 12x10 cluster map is produced as a.png file.



```python
def cluster_map(self, correlation_matrix):
    sns.clustermap(
        correlation_matrix.dropna(axis=1, how="all").dropna(axis=0, how="all"),
        annot=True,  # Show the correlation values in each cell
        cmap="coolwarm",  # Color map
        figsize=(12, 10),
    )  # Size of the figure
    plt.title("Cluster Map of Features")
    plt.savefig(f"{PATH}/data/cluster_map.png")
```

count_plot

- The count_plot method creates a bar plot to display the frequency of the categorical variables. The Seaborn method count plot is used to generate the plot and it finally saves the plot as an image file.



```python
def count_plot(self, config, name, specific_name, y: str = "Frequency"):
    plt.figure(figsize=(12, 8))
    sns.countplot(**config)
    plt.title(name)
    plt.xlabel(specific_name)
    plt.ylabel(y)
    plt.xticks(rotation=45, ha="right")
    plt.tight_layout()
    plt.savefig(f"{PATH}/data/count_plot_{name}_{specific_name}.png")
```

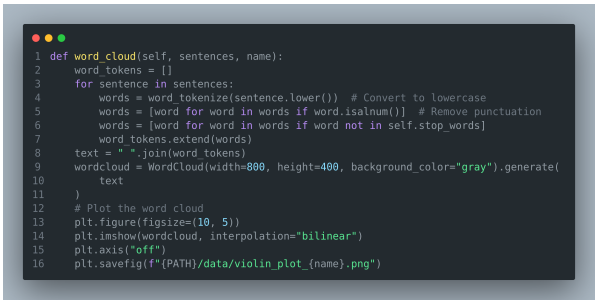
violin_plot

- The violin_plot method creates a violin plot to visualize the distribution of a numerical variable. This is implemented with the use of the violinplot method of Seasborn; the color palette is randomly chosen. Finally, the plot is saved to an image file.

```python
def violin_plot(
    self,
    title,
    x_name="specKey",
    y_name="movieYear",
):
    plt.figure(figsize=(12, 8))
    sns.violinplot(
        data=self.df,
        x=x_name,
        y=y_name,
        palette=f"Pastel{random.randint(1,2)}",
    )
    plt.title(title)
    plt.xlabel(x_name)
    plt.ylabel(y_name)
    plt.xticks(rotation=45, ha="right")
    plt.tight_layout()
    plt.savefig(f"{PATH}/data/violin_plot_{title}.png")

```

word_cloud

- The work_cloud method takes in a list of sentences and a name's parameters. It tokenizes each sentence, lowercases all of them, removes punctuation, and stops words, then combines them into a single text string. Then the combined text string is passed through to WordCloud and then it generates a visualization of the words. Finally, the visualization is saved as an image file.



```python
def word_cloud(self, sentences, name):
    word_tokens = []
    for sentence in sentences:
        words = word_tokenize(sentence.lower())  # Convert to lowercase
        words = [word for word in words if word.isalnum()]  # Remove punctuation
        words = [word for word in words if word not in self.stop_words]
        word_tokens.extend(words)
    text = " ".join(word_tokens)
    wordcloud = WordCloud(width=800, height=400, background_color="gray").generate(
        text
    )
    # Plot the word cloud
    plt.figure(figsize=(10, 5))
    plt.imshow(wordcloud, interpolation="bilinear")
    plt.axis("off")
    plt.savefig(f"{PATH}/data/violin_plot_{name}.png")
```

pair_plot

- The pair plot method takes in a list of attributes that are used as the columns and any rows without NaN values are dropped. With the use of Seaborn Pairplot functionality, the pair plot is automatically saved as an image file.

```python
def pair_plot(self, attributes):
    pp_df = self.df[attributes]
    pp_df.dropna(inplace=True)
    sns.pairplot(pp_df)
    plt.savefig(f"{PATH}/data/pair_plot.png")
```

analyze

- The analyze method is the main method that manages all of the other methods in the class. It first combines the data using the combine() method, then produces a correlation matrix, which is used for heatmaps and cluster maps. Then it presents the frequency of the technical specification using count_plot and finds the top 20 most common specification values. Violin plots are then used to find the distribution of technical specifications across different attributes. Additionally, word clouds are created for

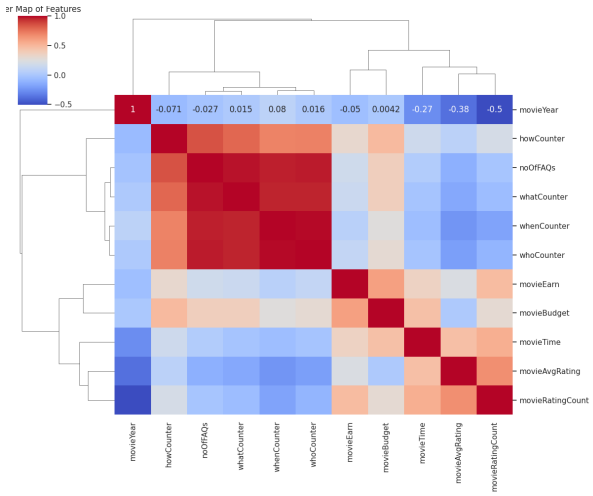
```python
def analyze(self):
    df = self.combine()
    correlation_matrix = self.correlation_matrix()
    self.heatmap(correlation_matrix=correlation_matrix)
    self.cluster_map(correlation_matrix=correlation_matrix)
    self.count_plot(
        {"data": df, "x": "specKey", "palette": "viridis"},
        "Frequency of Technical Specifications",
        "Specification Key",
    )
    self.count_plot(
        {
            "data": df,
            "y": "specValue",
            "palette": "magma",
            "order": self.df["specValue"].value_counts().index[:20],
        },
        "Top 20 Most Common Specification Values",
        "Frequency",
        "Specification Key",
    )
    self.violin_plot(
        "Distribution of specKey by movieYear",
        "specKey",
        "movieYear",
    )
    self.violin_plot(
        "Distribution of specKey by movieEarn",
        "specKey",
        "movieEarn",
    )
    self.violin_plot(
        "Distribution of specKey by movieAvgRating",
        "specKey",
        "movieAvgRating",
    )
    self.violin_plot(
        "Distribution of specKey by movieTime",
        "specKey",
        "movieTime",
    )
    self.word_cloud(self.descriptions.dropna(), "Descriptions")
    self.word_cloud(self.story_lines.dropna(), "StoryLines")
    self.pair_plot(
        [
            "movieYear",
            "movieTime",
            "movieAvgRating",
            "movieRatingCount",
            "movieEarn",
            "movieBudget",
        ]
    )
```

movie descriptions and storylines. Finally, a pair plot is created to show the pariwise relationships among the numeric attributes.

Analysis Explanation

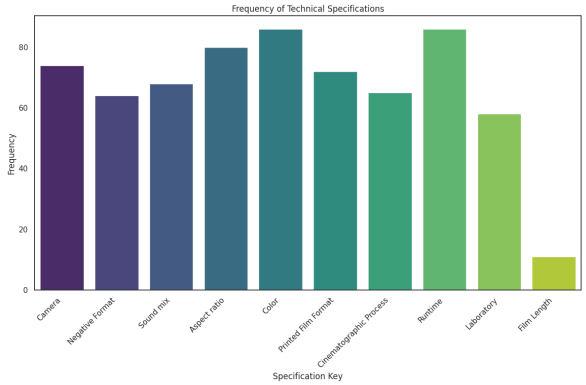
Cluster Map

- Shows groups of movies that have been automatically clustered together based on shared features



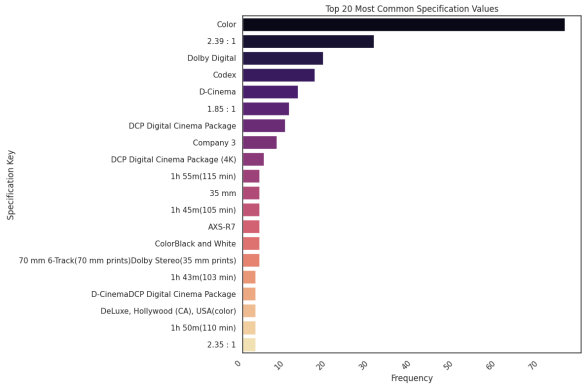
count_plot_Frequency of Technical Specifications_Specification Key

- The bar chart shows the number of times each technical specification key appears in the data set.



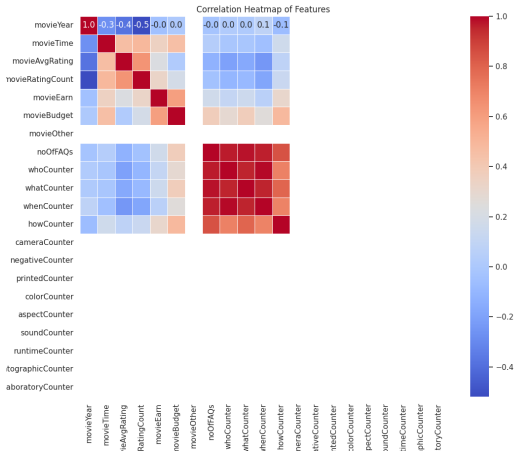
count_plot_Top 20 Most Common Specification Values_Frequency

- Shows the most frequent values for technical specifications.



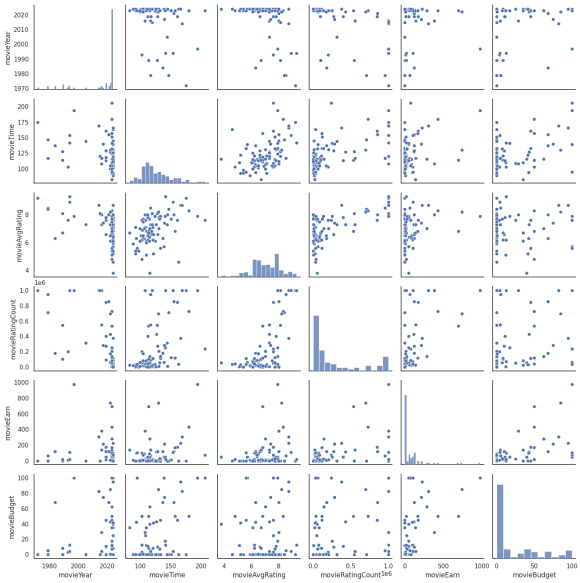
Heat Map

- The heatmap visualizes the correlation between different movie features.



Pair Plot

- The plot shows the relationship between various movie properties. Each square chart shows the distribution of a pair of movie properties.

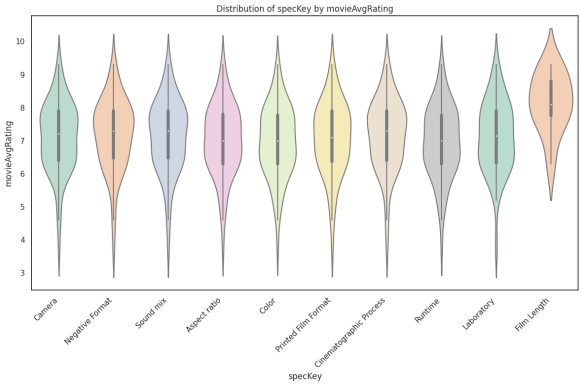


Word Usage Descriptions

- The word cloud shows that the most common words used in descriptions of the movies

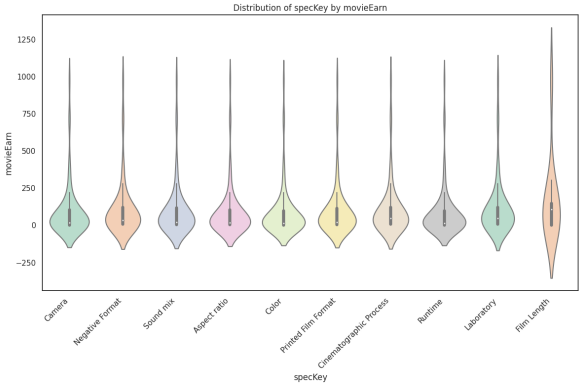
violin_plot_Distribution of specKey by movieAvgRating

- The violin plot visualizes the distribution of movie specifications (specKey) according to the average movie rating (movieAvgRating). The wider spread of the violin plot indicates a greater range of specifications for movies with that average rating.



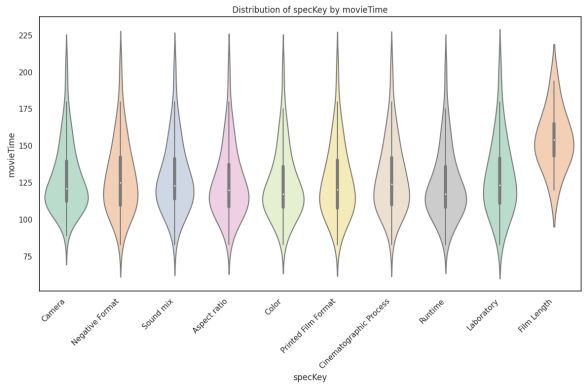
violin_plot_Distribution of specKey by movieEarn

- The violin plot you sent shows the distribution of movie specifications (specKey) by movie earnings (movieEarn). The wider part of the violin body represents the concentration of movies with those specifications.



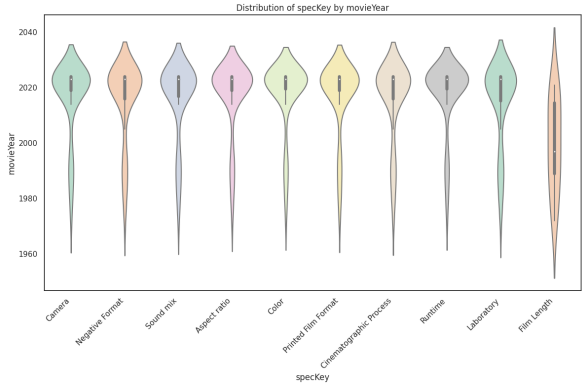
violin_plot_Distribution of specKey by movieTime

- The violin plot you sent shows the distribution of movie specifications (specKey) by movie runtime (movieTime). The wider areas of the violin plots represent the most common movie runtime lengths for movies with those specifications.



violin_plot_Distribution of specKey by movieYear

- The violin plot you sent shows the distribution of movie specifications (specKey) by movie release year (movieYear). The wider areas of the violin plots represent the most common release years for movies with those specifications.



Word Usage Story Lines

- The most prominent words in the movies.

Insights and Future Work

- Scraping Additional Data Points: Scraping more data points, such as user reviews, critical reviews, and user comments, will help provide a more thorough analysis and understanding of the data.
- Exploring Advanced Modeling Techniques: Techniques such as Natural Language Processing (NLP) can be used to get the sentiment of the text's user and allow for additional analytical data points.
- Dynamic visualization: It would be easier and much more productive if the visualization were dynamic and interactive, allowing for an overall better experience.

References

Lewis, R. (2024), *IMDB: History, Features, & Facts*. https://www.britannica.com/topic/IMDb.

Chhikara, M. (2022), 'What is the Internet Movie Database (IMDb)? All You Need to Know,' *Jagranjosh.com*, 12 October. https://www.jagranjosh.com/general-knowledge/what-is-internet-movie-database-imdb-all-you-need-to-know-1665565087-1.

How to Scrape Movie Data from IMDb (no date). https://crawlbase.com/blog/scrape-imdb-movie-data/.

Fisher, S. (2022) *What is IMDB?* https://www.lifewire.com/internet-movie-database-3482288.

codedamn news. (2024). *Selenium vs. Beautiful Soup: What is the difference?* [online] Available at: https://codedamn.com/news/selenium/selenium-vs-beautifulsoup-what-is-the-difference [Accessed 29 Apr. 2024].

kaggle.com. (n.d.). *Web scraping using BeautifulSoup*. [online] Available at: https://www.kaggle.com/code/ayush10mehta/web-scrapping-using-beautifulsoup [Accessed 29 Apr. 2024].

Python, R. (n.d.). *Python’s Requests Library (Guide)–Real Python*. [online] realpython.com. Available at: https://realpython.com/python-requests/.

Kim, K. (2023). *Python's Requests Module: A Comprehensive Guide*. [online] Medium. Available at: https://medium.com/@kevinkoech265/pythons-requests-module-a-comprehensive-guide-77b8acd1b15f.