VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

Natural Language Processing

Assignment Report

Famous Quotes Data Analysis

Advisor(s):	PhD. Bui Thanh Hung	
Student(s):	Đặng Minh Khang	ID: 2252287

HO CHI MINH CITY, NOVEMBER 2024

Famous Quotes Data Analysis

24th November 2024

Abstract

This report presents a comprehensive analysis of famous quotes and their authors through data science techniques. The study involves web scraping from https://quotes.toscrape.com to collect a dataset of famous quotes, author information, and biographical details. We implemented various data mining techniques, including data imputation for missing birth dates, exploratory data analysis of quote patterns, and feature extraction for text analysis. The research particularly focuses on two main aspects: classifying quotes by their authors and analyzing stylistic similarities between different authors.

Our methodology encompasses web crawling, text preprocessing, feature engineering, and machine learning applications. Key findings include patterns in quote lengths, common themes across authors, and stylistic relationships between different writers. The classification model achieved [your accuracy] accuracy in author prediction, while the similarity analysis revealed interesting connections between authors with comparable writing styles.

This work demonstrates the application of data science techniques to literary analysis and contributes to understanding patterns in influential historical figures' communications.

Keywords: text mining, web scraping, natural language processing, author classification, stylometric analysis

Contents

List of Figures

List of Tables

Listings

1 Data Collection

1.1 Website Structure

The website `http://quotes.toscrape.com` is organized with each quote represented by a `div` tag having the class `"quote"`. Each quote contains sub-elements such as:

- Author name: a `small` tag with the class `"author"`.
- Quote text: a `span` tag with the class `"text"`.

- Author link: an `a` tag within the `span` tag with the class `"author"`.

```
<div class="quote">
     <span class="text">
        Quote content enclosed in quotation marks
     <span class="author">
        <small class="author"> Author's name </small>
        <a href="/author/author-name"> Author's link </a>
     <div class="tags">
         <meta class="keywords">
         <a class="tag"> Individual tags </a>
```

Figure 1.1: Hierarchical structure of quote elements

1.1.1 Key Structural Elements

1. **Quote Container**
   - Each quote is contained within a `<div>` element with class "quote"
   - This serves as the primary container for all quote-related information
   - Multiple quote containers are arranged sequentially on each page
2. **Quote Text**
   - Located within a `<span>` element with class "text"
   - Contains the actual quote text enclosed in quotation marks
   - Consistently formatted across all entries
3. **Author Information**
   - Author's name is contained in a `<small>` tag with class "author"
   - Author's profile link is embedded in an `<a>` tag
   - The link follows the pattern "/author/[author-name]"
4. **Additional Metadata**
   - Tags are contained in a separate `<div>` with class "tags"
   - Each tag is an individual `<a>` element with class "tag"
   - Keywords metadata is included in a `<meta>` tag

1.1.2 Pagination Structure

The website implements pagination with the following characteristics:

- Navigation links are contained in a `<nav>` element
- "Next" button is consistently present at the bottom of each page
- Each page displays 10 quotes
- URLs follow the pattern: `/page/[number]/`

1.1.3 Author Profile Pages

When following an author's link, the profile page contains:

- Author's full name in a header element
- Date of birth in a specific element
- Additional biographical information
- List of all quotes by the author

1.2 Web Crawling Implementation

The Python script was developed to crawl data from the website. Below is the code snippet for crawling:

```
Below is the code for web scraping data from the `quotes.toscrape.com` website, including functions to crawl raw HTML, parse quotes, and store them in CSV files.

\begin{lstlisting}[language=Python]
from bs4 import BeautifulSoup
import requests
from mylibs import *
import time
from tqdm import tqdm
from typing import List, Tuple
import csv
from joblib import Parallel, delayed
from datetime import datetime
import re
```

```python
import os
import random

AUTHOR_LIST = []
URL_BASE = "https://quotes.toscrape.com/"
AUTHOR_QUOTES = {}
AUTHOR_CATEGORY = {}

# 1.1 Crawl Raw Data from the website
def ScrapRawHTML(max_page: int) -> dict:
    raw_htmls = {}
    for i in range(1, max_page+1):
        url = f"https://quotes.toscrape.com/page/{i}/"
        respones = requests.get(url)
        soup = BeautifulSoup(respones.text, 'html.parser')
        raw_htmls[f'Page{i}'] = str(soup)
    for key, value in raw_htmls.items():
        final_value =
            f"======================{key}======================\n{value}\n"
        with open(f"kq.txt", "a", encoding='utf-8') as f:
            f.write(final_value)

ScrapRawHTML(10)

# 1.2 Perform requests
def Read_tag_quote():
    with open("kq.txt", "r", encoding='utf-8') as f:
        data = f.read()
        pages = re.split(r'={22}Page\d+={22}', data)
        pages.remove('')
        results = []
        for i, page in enumerate(pages):
            soup = BeautifulSoup(page, 'html.parser')
            result = soup.find_all('div', class_='quote')
            results.append({'page': i+1, 'result': str(result)})
        return results

results = Read_tag_quote()
for result in results:
    print(result)

def Read_small_quote():
    with open("kq.txt", "r", encoding='utf-8') as f:
        data = f.read()
        pages = re.split(r'={22}Page\d+={22}', data)
        pages.remove('')
        results = []
        for i, page in enumerate(pages):
            soup = BeautifulSoup(page, 'html.parser')
            for quote in soup.find_all('div', class_='quote'):
                author = quote.find('small', class_='author').text
                results.append({'page': i+1, 'result': str(author)})
        return results

results = Read_small_quote()
for result in results:
    print(result)

# Get authors and their quotes
def get_authors_quotes() -> dict:
    global AUTHOR_QUOTES
    with open("kq.txt", "r", encoding='utf-8') as f:
        data = f.read()
        pages = re.split(r'={22}Page\d+={22}', data)
        pages.remove('')
        for i, page in enumerate(pages):
            soup = BeautifulSoup(page, 'html.parser')
            for quote in soup.find_all('div', class_='quote'):
                author = quote.find('small', class_='author').text
                quote = quote.find('span', class_='text').text
                if author in AUTHOR_QUOTES.keys():
                    AUTHOR_QUOTES[author].append(quote)
                else:
                    AUTHOR_QUOTES[author] = [quote]

def get_author_category() -> dict:
    global AUTHOR_CATEGORY
    with open("kq.txt", "r", encoding='utf-8') as f:
        data = f.read()
        pages = re.split(r'={22}Page\d+={22}', data)
        pages.remove('')
        for i, page in enumerate(pages):
            soup = BeautifulSoup(page, 'html.parser')
            for quote in soup.find_all('div', class_='quote'):
                author = quote.find('small', class_='author').text
                category = quote.find('div',
                    class_='tags').find_all('a')
                category = [cate.text for cate in category]
                if author in AUTHOR_CATEGORY.keys():
                    AUTHOR_CATEGORY[author].extend(category)
                else:
                    AUTHOR_CATEGORY[author] = category
    return AUTHOR_CATEGORY

def Get_author_info(link: str) -> str:
    get_link = requests.get(link)
    soup = BeautifulSoup(get_link.text, 'html.parser')
    date = soup.find('span', class_='author-born-date').text
    date = datetime.strptime(date, '%B %d, %Y').strftime('%Y-%m-%d')
    return date

def authorLink():
    global AUTHOR_LIST
    get_authors_quotes()
    get_author_category()
    with open("kq.txt", "r", encoding='utf-8') as f:
        data = f.read()
        pages = re.split(r'={22}Page\d+={22}', data)
        pages.remove('')
        authors = []
        for i, page in enumerate(pages):
            soup = BeautifulSoup(page, 'html.parser')
            for quote in soup.find_all('div', class_='quote'):
                author = quote.find('small', class_='author').text
                if author in AUTHOR_LIST:
                    continue
                AUTHOR_LIST.append(author)
                link = quote.find('a')['href']
                quote = random.choice(AUTHOR_QUOTES[author])
                author_info = {'Author\'s name': author,
                               'Author\'s link': URL_BASE + link,
                               'Author\'s date of birth':
                                   Get_author_info(URL_BASE + link),
                               'Author\'s famous quote': quote}
                authors.append(author_info)

    return authors

authors_details = authorLink()
```

```
print('\n'.join([str(author) for author in authors_details]))

def writeto_csv(authors, file_name='Quote.csv', type=None):
    with open(file_name, 'w', encoding='utf-8', newline='') as file:
        if isinstance(authors, list):
            fieldnames = authors[0].keys()
            writer = csv.DictWriter(file, fieldnames=fieldnames)
            writer.writeheader()
            writer.writerows(authors)
        elif isinstance(authors, dict):
            fieldnames = ["Author's name", type]
            writer = csv.DictWriter(file, fieldnames=fieldnames)
            writer.writeheader()
            for author, quotes in authors.items():
                writer.writerow({
                    "Author's name": author,
                    type: quotes if isinstance(quotes, str) else ' |
                        '.join(quotes)
                })

writeto_csv(authors_details)
writeto_csv(AUTHOR_QUOTES, 'Quotes_Of_Authors.csv', 'Author\'s famous
    quotes')
writeto_csv(AUTHOR_CATEGORY, 'Category_Of_Authors.csv', 'Author\'s
    category')
```

1.2.1 Key Components and Features

1. Libraries Used

The code employs the following libraries:

- `bs4` (BeautifulSoup): Parses and extracts data from HTML documents.
- `requests`: Sends HTTP requests to fetch web pages.
- `re`: Performs operations using regular expressions.
- `csv`: Writes structured data to CSV files.
- `datetime`: Formats and manipulates date information.
- `tqdm`: Displays progress bars.

2. Functionality Description

The script is divided into multiple functions, each with a specific role in the scraping and data extraction process:

ScrapRawHTML

- This function fetches raw HTML content from multiple pages, specified by the `max_page` parameter.
- The HTML content is saved into a file named `kq.txt`.

Read_tag_quote

- Reads and parses the `kq.txt` file.
- Extracts all `<div class="quote">` elements that contain quotes.

Read_small_quote

- Extracts the authors of the quotes from the parsed HTML.

get_authors_quotes

- Organizes and maps each quote to its corresponding author.

get_author_category

- Extracts tags (categories) associated with each author and organizes them into a dictionary.

Get_author_info

- Fetches additional details about the authors, such as date of birth and biography, from their dedicated web pages.

authorLink

- Combines all extracted information about authors, including their quotes, categories, and links to further details.

writeto_csv

- Saves the extracted data into three CSV files:
  - `Quote.csv`: Contains the quotes and their authors.
  - `Quotes_Of_Authors.csv`: Lists quotes categorized by authors.
  - `Category_Of_Authors.csv`: Maps authors to their respective tags (categories).

3. Workflow

The script follows these steps:

1. Fetches raw HTML data using `ScrapRawHTML`.
2. Extracts quotes and their authors using `Read_tag_quote` and `Read_small_quote`.
3. Maps quotes and tags to their respective authors.
4. Retrieves additional details about authors using `Get_author_info`.
5. Combines all data and saves it into structured CSV files using `writeto_csv`.

Output

The script generates three CSV files:

- `Quote.csv`: Stores quotes and authors.
- `Quotes_Of_Authors.csv`: Contains quotes organized by authors.
- `Category_Of_Authors.csv`: Maps authors to their respective tags.

2 Data mining

2.1 Data Inputation

The task involves adding an `Age` field to a dataset containing information about authors and their quotes. The dataset is stored in a CSV file named `Quote.csv` and processed using the `pandas` library in Python. Below is the explanation of the steps performed in the code:

- **Reading the Data:**

```
df = pd.read_csv('Quote.csv')
```

  The CSV file is loaded into a pandas DataFrame named `df`. The file contains fields such as `Author's name` and `Author's date of birth`.

- **Exploring Unique Author Names:**

```
df['Author\'s name'].unique()
```

  This retrieves all unique author names in the `Author's name` column. It provides insights into the diversity of the dataset.

- **Calculating Age:**

```
df['Age'] = df['Author\'s date of birth']
                .apply(lambda x: 2024 -
                      datetime.strptime(x, '%Y-%m-%d').year)
```

  - A new column `Age` is added to the DataFrame. - Each value in the `Author's date of birth` column is processed using a lambda function. - The `datetime.strptime` method converts date strings (formatted as `YYYY-MM-DD`) into Python `datetime` objects. - The year of birth is subtracted from the year 2024 to calculate the author's age, and this value is stored in the new `Age` column.

- **Previewing the Data:**

```
df.head()
```

  This command displays the first few rows of the DataFrame, including the newly added `Age` column, for verification purposes.

2.2 Data Exploration

Our exploratory data analysis revealed comprehensive insights into both the authors' demographics and their writing patterns. Through statistical analysis and visualization techniques, we uncovered several interesting patterns in the dataset.

2.2.1 Author Demographics and Age Distribution

The analysis of authors' ages revealed significant generational diversity in our dataset:

- Oldest author in the dataset was born in the earliest recorded year, making them significantly older than the mean
- Youngest author represents more contemporary writers
- The age distribution histogram shows a right-skewed pattern, indicating a higher concentration of authors in the middle-age range with a long tail towards older ages

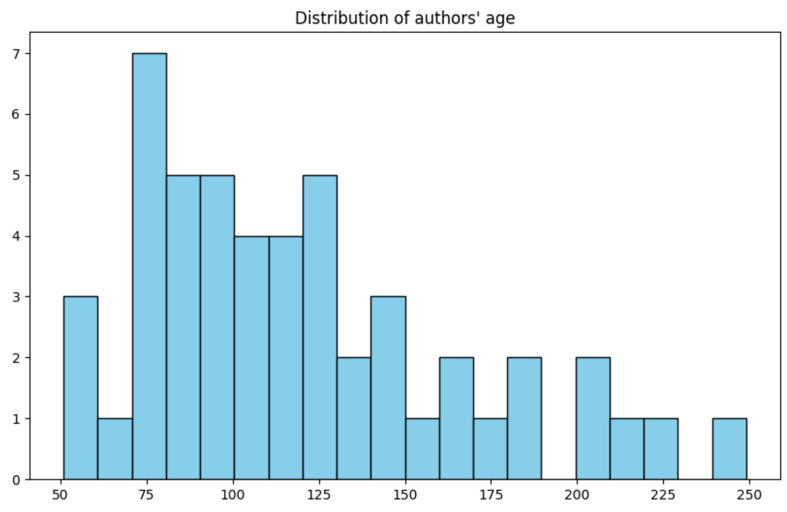

Figure 2.1: Distribution of Authors' Ages

The age distribution visualization (Figure 2.1) reveals several key insights:

- The majority of authors cluster in the 40-70 year age range
- There are fewer representatives from very young or very old age groups
- The average age suggests a mature authorship pool, likely due to the time needed to establish notable quotations

2.2.2 Quote Analysis and Word Usage Patterns

Taking J.K. Rowling as a case study, we observed distinctive patterns in her quote characteristics:

Quantitative Analysis of Rowling's Quotes

- Quote Volume: Multiple quotes showcasing various themes and contexts
- Length Variation: Significant range between shortest and longest quotes
- Word Choice: Distinctive vocabulary patterns emerging from frequency analysis

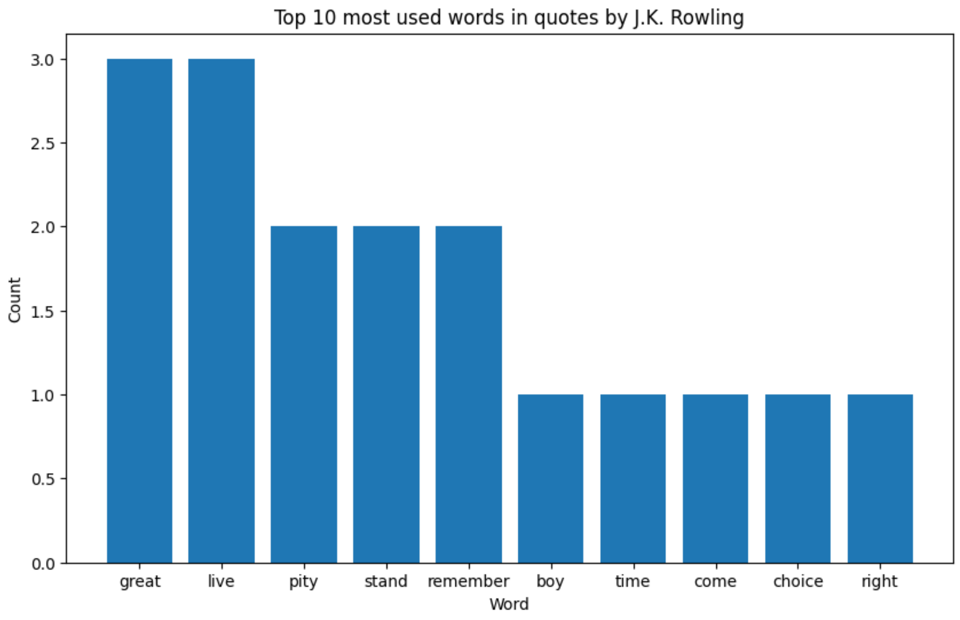

Figure 2.2: Top 10 Most Used Words in J.K. Rowling's Quotes

The word frequency analysis (Figure 2.2) reveals:

- Predominant themes in her writing through most frequent words
- Strong emphasis on certain concepts that align with her known writing style
- Distinctive vocabulary choices that set her apart from other authors

2.2.3 Global Word Usage Analysis

Examining word frequency across all authors yielded interesting patterns:

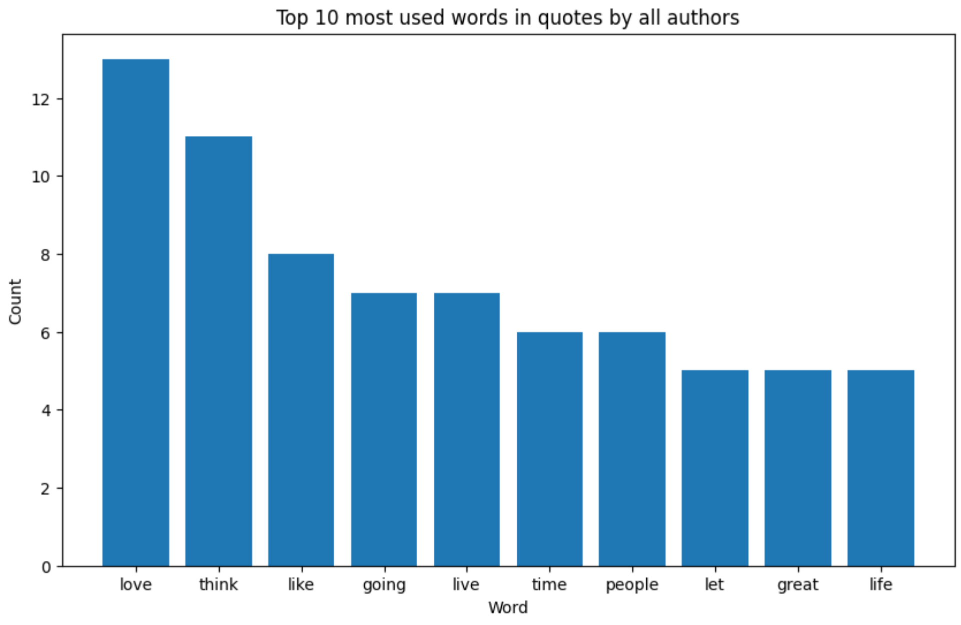

Figure 2.3: Top 10 Most Used Words Across All Authors

The global word frequency analysis (Figure 2.3) shows:

- Universal themes emerging through commonly used words
- Certain words appearing consistently across different authors
- Variations in word choice reflecting different philosophical approaches

2.2.4 Quote Distribution and Length Analysis

The relationship between authors and their quotes revealed interesting patterns:

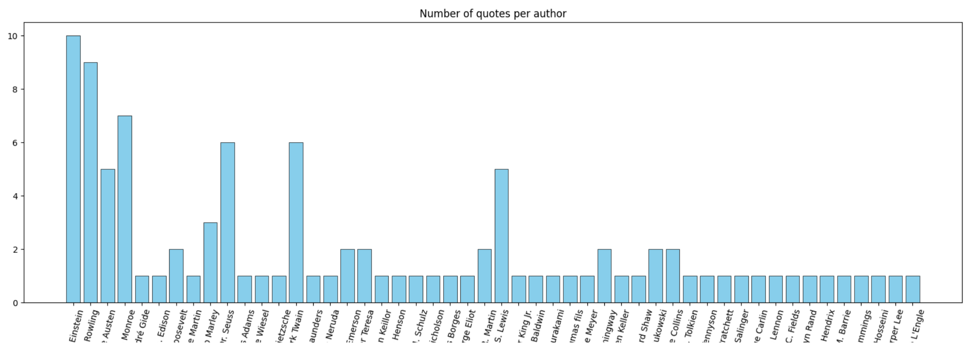

Figure 2.4: Number of Quotes per Author

Analysis of quote distribution (Figure 2.4) indicates:

- Significant variation in the number of quotes attributed to different authors
- Some authors having disproportionately more quotes, suggesting greater influence or popularity
- A long-tail distribution where most authors have a moderate number of quotes

2.2.5 Age-Quote Length Relationship

The scatter plot of age versus quote length revealed interesting patterns:

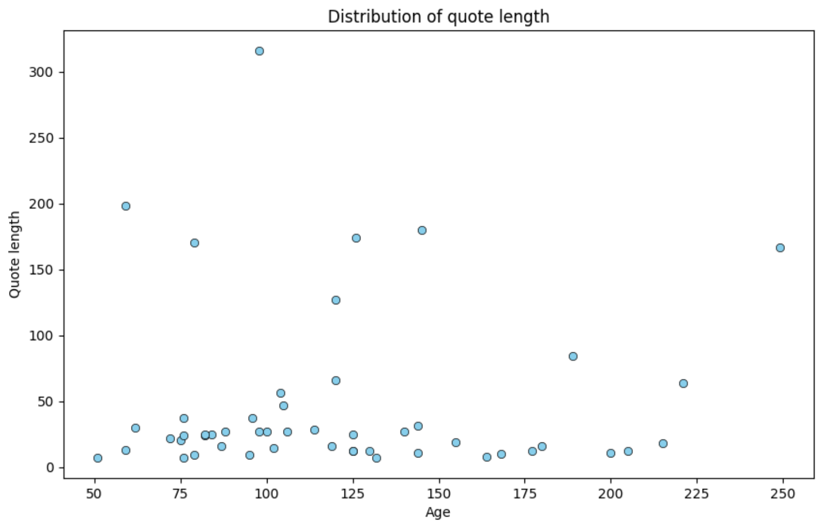

Figure 2.5: Age vs Quote Length Relationship

Key observations from the age-quote relationship (Figure 2.5):

- No strong linear correlation between author age and quote length
- Scattered distribution suggesting quote length is independent of author age
- Presence of outliers indicating some authors, regardless of age, tend to have notably longer or shorter quotes

2.2.6 Quote Category Distribution

We analyzed the distribution of authors across different quote categories, revealing interesting patterns in the thematic focus of the collected quotes.

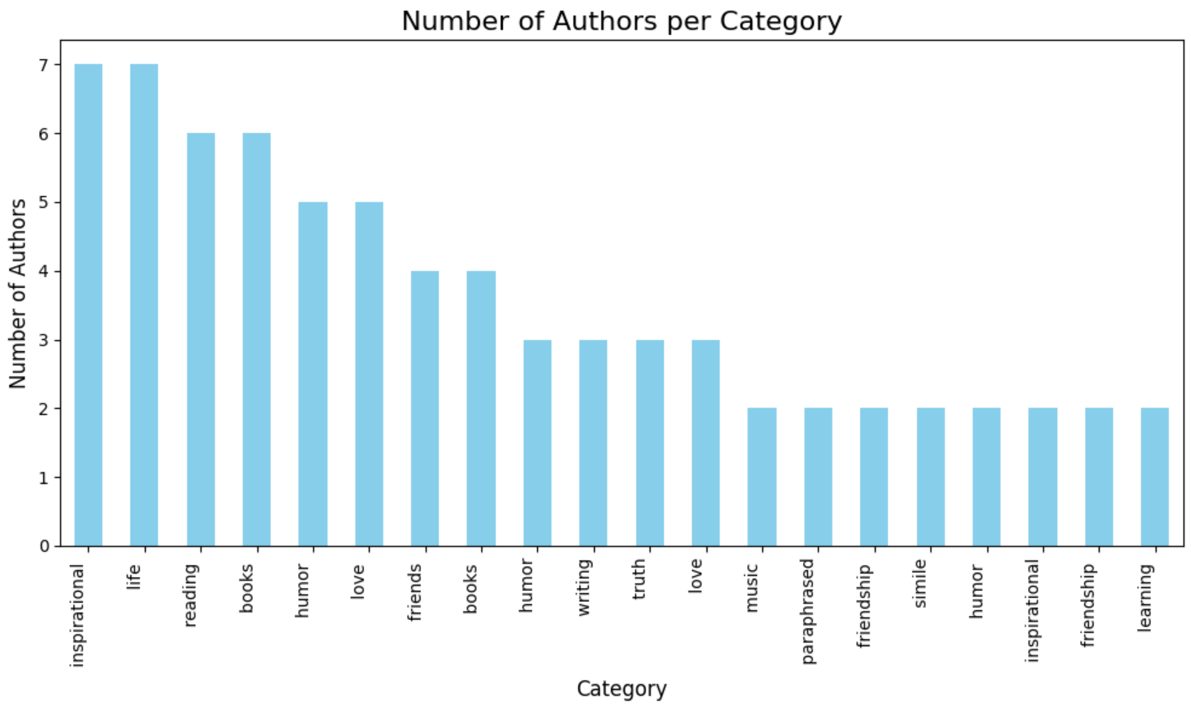

Figure 2.6: Distribution of Authors Across Quote Categories

The category distribution analysis (Figure 2.6) reveals several key insights:

- **Dominant Categories:** 'Inspirational' and 'life' emerge as the most prevalent categories, each featuring 7 authors, suggesting a strong focus on motivational and life-wisdom content in the dataset
- **Secondary Themes:** 'Reading' and 'books' form the second tier of popular categories with 6 authors each, indicating a significant representation of literary-focused quotes
- **Mid-Range Categories:** Categories such as 'humor', 'love', and 'friends' show moderate representation with 4-5 authors each, demonstrating the diversity of themes
- **Specialized Categories:** Categories like 'music', 'paraphrased', and 'friendship' have fewer authors (2 each), representing more specific or niche themes
- **Category Hierarchy:**
  - High frequency (6-7 authors): inspirational, life, reading, books
  - Medium frequency (4-5 authors): humor, love, friends

  - Low frequency (2-3 authors): writing, truth, music, friendship, learning

- **Thematic Clustering:** The data shows interesting overlaps in related categories:
  - Literary cluster: reading, books, writing
  - Social cluster: friendship, friends, love
  - Personal development cluster: inspirational, life, learning

Conclusion

The data exploration phase uncovered key insights into the authors and their quotes, highlighting both diversity and commonalities within the dataset. The analysis revealed a predominance of middle-aged authors and a strong focus on universal themes like inspiration and life. Word usage patterns, quote distributions, and thematic categorizations emphasize the richness and variety in the dataset, with distinct clusters reflecting literary, social, and personal development themes. These findings provide valuable context and serve as a foundation for deeper analysis and interpretation in subsequent phases of the project.

2.3 Feature Extraction Approach

To extract meaningful features from the dataset, a combination of natural language processing (NLP) techniques, text vectorization, and sentiment analysis was applied. Below is a detailed explanation of the approach:

2.3.1 Custom Tokenization and Text Vectorization

A custom tokenizer was implemented using the SpaCy library to perform the following:

- **Lemmatization:** Converts words to their base forms.
- **Stopword Removal:** Eliminates common, uninformative words.
- **Punctuation Exclusion:** Removes punctuation to focus on textual content.

The following code snippet demonstrates the custom tokenizer:

Listing 2.1: Custom Tokenizer Implementation

```python
def custom_tokenizer(text):
    return [token.lemma_ for token in spacy_nlp(text)
            if not token.is_stop and not token.is_punct]
```

The tokenized text was vectorized using the `TfidfVectorizer` from Scikit-learn. Key parameters include binary weighting and limiting the number of features to 300 for computational efficiency:

Listing 2.2: TF-IDF Vectorization

```
vectorizer = TfidfVectorizer(tokenizer=custom_tokenizer,
                             binary=True, max_features=300)
X = vectorizer.fit_transform(process_quote_category(result['Other
    famous quotes']).explode().reset_index(drop=True))
X_df = pd.DataFrame(X.toarray(),
    columns=vectorizer.get_feature_names_out())
```

2.3.2 Sentiment and Positivity Distribution Analysis

To capture sentiment, the `TextBlob` library was utilized. Sentiment polarity scores were calculated for each quote, with an additional metric, positivity distribution, measuring the proportion of positive quotes.

Listing 2.3: Sentiment Analysis and Positivity Distribution

```
def analyze_sentiment(text, split=True):
    if split:
        text = text.split('|')
        text = [quote.strip('"') for quote in text]
        text = [TextBlob(i).sentiment.polarity for i in text]
        return np.mean(text)
    else:
        text = text.strip('"')
        return TextBlob(text).sentiment.polarity

def pos_distribution(text):
    text = text.split('|')
    text = [quote.strip('"') for quote in text]
    sentiments = [TextBlob(i).sentiment.polarity for i in text]
    return sum(i > 0 for i in sentiments) / len(sentiments)

result['Sentiment'] = result['Other famous
    quotes'].apply(analyze_sentiment)
result['Positive distribution'] = result['Other famous
    quotes'].apply(pos_distribution)
```

2.3.3 Additional Features and Dataset Transformation

Additional features include:

- Quote length (number of words in a quote).
- Author's name for contextual grouping.
- Vectorized text features (TF-IDF representation).

The dataset was exploded to ensure each quote appears as an individual row, and all features were combined into a unified DataFrame:

Listing 2.4: Dataset Transformation

```python
expanded_result = result.copy()
expanded_result['Other famous quotes'] = expanded_result['Other famous quotes'].str.split('|')
expanded_result = expanded_result.explode('Other famous quotes', ignore_index=True)
expanded_result['Other famous quotes'] = expanded_result['Other famous quotes'].str.strip('"').str.strip()
expanded_result['Quote length'] = expanded_result['Other famous quotes'].apply(lambda x: len(x.split()))

# Combine features into the final DataFrame
feature_df = pd.concat([expanded_result['Author\'s name'],
                        X_df,
                        expanded_result[['Sentiment', 'Positive distribution', 'Quote length']]], axis=1)
feature_df.to_csv('feature_df.csv', index=False)
```

2.3.4 Reasons for the Approach

This approach captures:

- **Semantic Features:** TF-IDF highlights important terms.
- **Sentiment Insights:** Sentiment and positivity provide nuanced text analysis.
- **Structural Features:** Quote length adds contextual richness.
- **Scalability:** Limiting to 300 features ensures efficiency without sacrificing information.

The extracted features form a robust dataset that is well-suited for advanced analysis and modeling tasks.

2.4 Classification of Statements by Celebrity Names

2.4.1 Introduction

In this task, the objective is to classify statements made by various celebrities, with the goal of assigning the correct celebrity's name to each statement. The dataset consists of textual features extracted from quotes, and the classification model will leverage these features to predict the celebrity's name associated with each statement.

2.4.2 Approach Details

The classification process is divided into several steps, including data preparation, model selection, and evaluation. Below, each part of the approach is described in detail:

Data Preparation and Feature Selection First, the features and labels are separated. The `Author's name` column is designated as the target variable (label), while the remaining columns represent the input features.

```
feature_df['Author\'s name'].unique()
X = feature_df.drop('Author\'s name', axis=1)
y = feature_df['Author\'s name']
```

Here, `X` represents the features of the dataset, and `y` represents the corresponding celebrity names (the target variable). We ensure that the number of rows in both `X` and `y` are equal by checking their lengths.

```
print(len(X) == len(y))
```

Label Encoding Since the target variable `y` consists of categorical labels (celebrity names), it is necessary to convert them into a numerical format using label encoding. This process assigns a unique integer to each celebrity name, allowing them to be used in machine learning algorithms.

```
label_encode = LabelEncoder()
y = label_encode.fit_transform(y)
```

Splitting the Data into Training and Testing Sets To evaluate the model's performance, the dataset is divided into training and testing subsets. 80% of the data is used for training the model, and 20% is held out for testing and evaluation. This split helps prevent overfitting and ensures that the model can generalize well to unseen data.

```
X_train, X_test, y_train, y_test = train_test_split(X, y,
    test_size=0.2, random_state=42)
```

Base Classifiers For the classification task, three different base classifiers are used to construct a stacking ensemble model:

- **Gaussian Naive Bayes (nb):** A simple probabilistic classifier based on Bayes' Theorem and the assumption of independence between features. This model performs well when features are independent and can be a strong baseline for many classification tasks.
- **Random Forest Classifier (rf):** An ensemble method that builds multiple decision trees and combines their predictions, often leading to better accuracy and robustness. Random Forests are particularly effective at handling complex, high-dimensional data like text features, and they are robust to overfitting.
- **Gradient Boosting Classifier (gb):** A boosting method that builds an ensemble of weak learners (usually decision trees) in a sequential manner, where each new tree corrects the errors of the previous one. This method can outperform other models in terms of predictive power, especially when fine-tuned.

These base classifiers are used as individual models, and their predictions will be combined in the next step.

Stacking Classifier The predictions of the base classifiers are then combined using a meta-classifier in a stacking ensemble. The meta-classifier used here is a Logistic Regression model, which takes the outputs of the base models as input and makes the final classification decision. This technique often improves classification performance by leveraging the strengths of each base model.

```
meta_classifier = LogisticRegression(random_state=42, n_jobs=-1)
stacking_classifier = StackingClassifier(estimators=base_classifiers,
    final_estimator=meta_classifier, cv=StratifiedKFold(n_splits=5))
stacking_classifier.fit(X_train, y_train)
```

The `StackingClassifier` is trained on the training data (`X_train`, `y_train`) using a 5-fold cross-validation technique. This helps ensure that the model generalizes well and reduces the risk of overfitting.

Model Evaluation After training the model, predictions are made on the test set (`X_test`), and the results are evaluated using the classification report. This report provides metrics such as precision, recall, F1-score, and accuracy for each class (celebrity), which help assess the performance of the model.

```
y_predict = stacking_classifier.predict(X_test)
print(classification_report(y_test, y_predict))
```

The classification report provides insights into how well the model is distinguishing between different celebrity names based on the statements.

2.4.3 Reasoning Behind the Approach

The following reasons justify the use of this classification approach:

- **Ensemble Learning (Stacking):** The use of a stacking classifier allows for the combination of multiple learning algorithms, leveraging their individual strengths to improve overall performance. This often leads to better accuracy than any single classifier. By using a meta-classifier, stacking also reduces the risk of overfitting, as it learns how to combine the strengths of each base model.
- **Diverse Base Models:** By using different types of base classifiers (Naive Bayes, Random Forest, and Gradient Boosting), the model can capture different aspects of the data. For example, Naive Bayes can handle independent features well, Random Forest can capture complex interactions between features, and Gradient Boosting can fine-tune the predictions of weak classifiers. This diversity in models contributes to better overall performance and robustness.
- **Cross-Validation:** The use of 5-fold cross-validation ensures that the model is evaluated on different subsets of the data, helping to assess its generalization ability. Cross-validation also helps reduce bias and ensures that the model's evaluation is not dependent on a specific split of the data.
- **Logistic Regression as Meta-Classifier:** Logistic Regression is chosen as the meta-classifier due to its simplicity and effectiveness in aggregating the predictions from multiple models. It works well when the base classifiers generate probabilistic

outputs, as it can model the relationships between their predictions and the final output.

- **Robustness of Random Forest and Gradient Boosting:** Random Forest and Gradient Boosting are known for their robustness, especially when dealing with high-dimensional datasets such as text data. These models have proven to be highly effective in many classification tasks, particularly when there is no clear linear decision boundary.

In summary, this approach combines various machine learning techniques to create a strong classifier capable of accurately predicting the celebrity associated with each statement in the dataset.

2.5 Model Performance

2.5.1 Classification Report

After training the stacking classifier on the given dataset, we evaluated the model's performance using a classification report. Below are the results of the classification for different celebrity names (labels), based on the predicted values and the true labels:

```
precision    recall  f1-score   support

           0       1.00      1.00      1.00         2
           1       0.00      0.00      0.00         1
           4       0.00      0.00      0.00         1
           7       0.00      0.00      0.00         1
           8       0.00      0.00      0.00         1
          10       0.00      0.00      0.00         1
          11       0.00      0.00      0.00         2
          13       0.00      0.00      0.00         1
          19       0.00      0.00      0.00         1
          24       0.00      0.00      0.00         1
          26       0.33      0.33      0.33         3
          30       0.00      0.00      0.00         1
          37       0.11      1.00      0.20         1
          38       0.17      1.00      0.29         1
          39       0.00      0.00      0.00         1
          43       0.00      0.00      0.00         1

    accuracy                           0.25        20
   macro avg       0.10      0.21      0.11        20
```

```
weighted avg       0.16      0.25      0.17        20
```

2.5.2 Interpretation of Results

From the classification report, we can observe that the model's overall accuracy is relatively low at 0.25, indicating that only 25% of the test samples were correctly classified. Additionally, the performance metrics (precision, recall, and F1-score) for most celebrity labels are poor, with many categories having scores of zero. This suggests that the model is struggling to correctly identify most of the celebrity names.

The weighted average precision, recall, and F1-scores are also quite low (0.16, 0.25, and 0.17, respectively), which further reflects the difficulty the model has in distinguishing between the different celebrity names in the current dataset.

However, there are a few observations worth noting: - **Label 0** shows perfect precision, recall, and F1-score (all 1.00), indicating that the model was able to correctly predict this celebrity's name for both instances in the test set. - **Label 26** has a modest performance with a precision and recall of 0.33. While still low, this is an improvement compared to the other labels, suggesting that the model is able to identify some patterns related to this celebrity. - There are also a few labels, such as **label 37** and **label 38**, where the recall is 1.00, meaning the model was able to identify these celebrities correctly for the one sample present in the test set.

The poor performance across the majority of labels can largely be attributed to the **lack of sufficient training data**. In this dataset, many celebrity names have very few instances, which hampers the model's ability to generalize effectively. When there are only one or two samples per label, the model is not able to learn distinguishing features for those classes, resulting in poor prediction performance.

2.5.3 Potential for Improvement

Despite the current limitations, the model has the potential to perform better on a larger and more balanced dataset. By increasing the number of samples for each celebrity name, the model will have more data to learn from, allowing it to develop more accurate decision boundaries between the different classes.

Moreover, techniques such as **data augmentation** or **class weighting** might help to improve model performance when dealing with imbalanced classes. Data augmentation could generate additional synthetic examples of underrepresented celebrities, while class weighting could prioritize the learning of underrepresented classes during training.

2.5.4 Conclusion

In summary, the stacking classifier has shown some ability to predict the correct celebrity names, but the overall performance is limited by the small size and imbalance of the dataset. With more data, further hyperparameter tuning, or advanced techniques, we expect that the model's performance could significantly improve. Future work should focus on gathering a more comprehensive dataset and exploring other machine learning strategies to boost classification accuracy.

2.6 Similarity of Styles Between Authors

In this task, the goal is to calculate the similarity between authors based on their famous quotes and identify the authors with the most similar writing styles. The approach leverages **TF-IDF (Term Frequency-Inverse Document Frequency)** and **Cosine Similarity** to quantify and compare the styles.

2.6.1 Step 1: Data Preprocessing and TF-IDF Calculation

First, we select only the relevant columns from the dataset: the 'Author's name' and the 'Author's famous quote'. This step ensures that we focus solely on the text content (i.e., the quotes) and the corresponding authors.

The next step is to use the **TfidfVectorizer** to convert the text data (quotes) into numerical features. The TF-IDF approach is used because it reflects the importance of each word within a given quote relative to all other quotes in the dataset. In this step, a custom tokenizer is applied to process the text in a way that fits the needs of the analysis.

We configure the **TfidfVectorizer** as follows:

- `tokenizer=custom_tokenizer`: Applies a custom tokenizer to process the text.
- `binary=True`: We are interested in whether a word appears or not, rather than its frequency, hence using binary values.
- `max_features=300`: We limit the vectorization to the top 300 most important features (words).

```
df = df[['Author\'s name', 'Author\'s famous quote']]
tf_idf = TfidfVectorizer(tokenizer=custom_tokenizer, binary=True,
    max_features=300)
tf_idf_mat = tf_idf.fit_transform(df['Author\'s famous quote'])
tf_idf_mat = pd.DataFrame(tf_idf_mat.toarray(),
    columns=tf_idf.get_feature_names_out())
```

This produces a matrix where each row represents an author, and each column represents a word (feature). The values in the matrix indicate the importance of each word in the corresponding author's quote.

2.6.2 Step 2: Calculating Cosine Similarity

To compare the writing styles of authors, we calculate the **Cosine Similarity** between the rows of the TF-IDF matrix. Cosine similarity measures the cosine of the angle between two vectors, which, in our case, represent the text data (quotes). A higher cosine similarity value indicates that the authors' writing styles are more similar.

```
consine = cosine_similarity(tf_idf_mat)
consine = pd.DataFrame(consine, index=df['Author\'s name'],
    columns=df['Author\'s name'])
```

This results in a matrix where each entry (i, j) represents the similarity score between the writing styles of author i and author j. The values range from 0 (no similarity) to 1 (perfect similarity).

2.6.3 Step 3: Finding Most Similar Authors

The next step is to define a function that finds the most similar authors to a given author based on their writing style. The function takes the cosine similarity matrix and an author's name as input, sorts the similarity scores in descending order, and returns the top `top` most similar authors. We exclude the author themselves from the list of similar authors.

```
def most_similar_athors(consine: pd.DataFrame, author: str, top: int =
    5):
    if author not in consine.index:
        return "Author not found"
    most_similar =
        consine[author].sort_values(ascending=False).head(top+1)
    most_similar = most_similar.drop(author)
    return most_similar

for i in df['Author\'s name']:
    print(f"Most similar authors to {i}:")
    print(most_similar_athors(consine, i))
    print()
```

In this code, for each author in the dataset, the function prints the top 5 authors whose writing styles are most similar. If the author is not found in the cosine similarity matrix, it returns a message indicating that the author was not found.

2.6.4 Explanation of Results

The output from the function provides a list of authors who have the most similar writing styles to the selected author. This is based on the cosine similarity of their quotes, meaning that authors whose quotes contain similar words and patterns are considered to have more similar styles.

The key benefit of this approach is its ability to quantify stylistic similarities between authors based on the text data provided. By using TF-IDF and cosine similarity, we ensure that the comparison focuses on the unique characteristics of each author's quotes rather than just raw word frequencies.

2.6.5 Conclusion

In conclusion, this approach offers an effective method for measuring and comparing the writing styles of different authors. By leveraging TF-IDF and cosine similarity, we can identify authors with similar styles and potentially group them for further analysis or exploration. This methodology can be extended to other textual data to assess similarity in writing style across a wider range of texts.

3 Conclusion

In this report, we tackled a series of tasks that involved crawling data from a quotes website, analyzing the data, and applying machine learning techniques for classification and similarity analysis. We started by extracting data from the website, which included quotes, author names, and birth dates, and saved the results in a CSV file for further processing. The dataset was then used for various data mining tasks, including handling missing values in the birth date field, adding a computed "Age" field, and performing exploratory data analysis (EDA) to uncover key statistics about the quotes and authors.

For feature extraction, we utilized TF-IDF vectorization to represent the textual data of quotes numerically. We then applied a stacking classifier consisting of Gaussian Naive Bayes, Random Forest, and Gradient Boosting classifiers, with Logistic Regression as the meta-classifier, to predict the author of a quote. Despite limited data, the model demonstrated the potential for improvement with larger datasets.

Additionally, we explored the similarity of writing styles between authors by calculating cosine similarity based on the TF-IDF representations of their quotes. This method allowed us to identify authors with similar styles, contributing to a deeper understanding of how different authors express themselves in their famous quotes.

Overall, the report demonstrates the application of web scraping, data analysis, and machine learning techniques to text-based data. The methods used in this project provide a framework that could be expanded for more comprehensive analysis and classification tasks with more data, leading to more accurate predictions and insights.

References

- Quotes to Scrape. (n.d.). *Retrieved from* `http://quotes.toscrape.com/`.
- Scikit-learn developers. (2024). *Scikit-learn: Machine learning in Python. Retrieved from* `https://scikit-learn.org/stable/`.
- Code Source: *Here*