anime_rs_final

March 2, 2025

1 The Anime Reccommendation Engine

1.1 Project Description:

This project aims to build an **anime recommendation system** using **unsupervised learning**. The dataset consists of anime dialogues, and the goal is to recommend similar dialogues using Natural Language Processing (NLP) techniques.

1.2 GitHub Link: https://github.com/kmora2b/anime_rs_project

1.3 Why?

Finding anime based on existing databases like MyAnimeList(MAL) and AniList often require a viewer to know specific genres, studios, and styles to even search for reccommendations but often this leads to a viewer to miss out on new and retro underrated shows or get misled. Sometimes being misled could lead to users watching something that can be uncomfortable or a lot more mature than they intended. With this reccommender, we will be able to know the direct contents of a show based on subtitles that contain dialogue of a show which will provide a better representation and allow a viewer to just get a reccommendation based on general keywords.

1.3.1 Key Steps:

1. **Exploratory Data Analysis (EDA)**:
   - Understanding dataset size, structure, and distributions.
   - Generating statistical insights and visualizations.
   - Identifying missing or redundant data.
   - Investigating text length distributions, common words, and stopword frequency.
   - Checking for imbalanced clusters and sparsity in TF-IDF features.
   - **Describing dataset factors (features)**, their importance, and distributions.
   - **Analyzing correlations between factors and determining dependencies.**
   - **Checking if data transformations (e.g., log transformation, normalization) are required.**
   - **Detecting and handling outliers or missing data in dialogues.**
2. **Preprocessing & Feature Extraction**: -NLP Based Cleaning
   - Cleaning text data (removing special characters, stopwords, etc.).
   - Tokenization and word frequency analysis.

   - Converting text into numerical representations using **TF-IDF (Term Frequency-Inverse Document Frequency)**.
   - Evaluating feature distributions and sparsity levels.
3. **Clustering using Multiple Models and Evaluation**:
   - Applying **K-Means clustering** to group similar anime dialogues.
   - Applying **Hierarchical Clustering** to compare performance.
   - Applying **DBSCAN (Density-Based Spatial Clustering)** to detect outlier dialogues.
   - Analyzing cluster results and interpreting themes.
   - Reducing dimensions using **Principal Component Analysis (PCA)** for visualization.
   - **Comparing clustering models using evaluation metrics**.
4. **Building the Recommendation System**:
   - Using **cosine similarity** to find similar dialogues.
   - Evaluating different similarity measures (e.g., Jaccard similarity, Euclidean distance).
   - Recommending anime dialogues based on textual similarity.
   - Implementing evaluation metrics to assess recommendation quality.
   - **Allowing user text input to search for specific anime-related keywords and recommending relevant anime**.
   - **Visualizing similarity scores for better interpretability**.
5. **Discussion & Conclusion**:
   - Evaluating the effectiveness of clustering and recommendations.
   - Identifying limitations and suggesting future improvements.
   - Addressing potential refinements and optimizations.
   - Exploring future methodologies such as deep learning models for text similarity.

[1]:

```python
# Import necessary libraries
import numpy as np
import pandas as pd
import json
import re
import os
import random
import zipfile
import matplotlib.pyplot as plt
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.cluster import KMeans, AgglomerativeClustering, DBSCAN,␣
 ↪MiniBatchKMeans
from sklearn.decomposition import PCA, TruncatedSVD
from sklearn.metrics.pairwise import cosine_similarity, euclidean_distances,␣
 ↪pairwise_distances
from sklearn.metrics import silhouette_score, davies_bouldin_score,␣
 ↪adjusted_rand_score, jaccard_score
from collections import Counter
import seaborn as sns
from sklearn.feature_extraction.text import ENGLISH_STOP_WORDS

# NLP Libraries
```

```python
!pip install nltk
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from nltk.stem import WordNetLemmatizer

# Download required NLP resources
nltk.download("stopwords")
nltk.download("punkt")
nltk.download("wordnet")
```

```
Requirement already satisfied: nltk in /opt/conda/lib/python3.7/site-packages
(3.8.1)
Requirement already satisfied: click in /opt/conda/lib/python3.7/site-packages
(from nltk) (7.1.2)
Requirement already satisfied: joblib in /opt/conda/lib/python3.7/site-packages
(from nltk) (0.14.1)
Requirement already satisfied: regex>=2021.8.3 in /opt/conda/lib/python3.7/site-
packages (from nltk) (2024.4.16)
Requirement already satisfied: tqdm in /opt/conda/lib/python3.7/site-packages
(from nltk) (4.45.0)
```

```
[nltk_data] Downloading package stopwords to /home/jovyan/nltk_data…
[nltk_data]   Package stopwords is already up-to-date!
[nltk_data] Downloading package punkt to /home/jovyan/nltk_data…
[nltk_data]   Package punkt is already up-to-date!
[nltk_data] Downloading package wordnet to /home/jovyan/nltk_data…
[nltk_data]   Package wordnet is already up-to-date!
```

```
[1]: True
```

1.4 Dataset Overview:

- **Citation Source**: J. F. "Anime Subtitles Dataset," Kaggle, 2023. Available: https://www.kaggle.com/datasets/jef1056/anime-subtitles.
- **Size**: Contains thousands of dialogue entries across multiple files.
- **Format**: Extracted from structured `.json` files with `context-response` pairs.
- **Data Types**:
  - **Text**: Contains natural language sentences from anime scripts.
  - **Features Extracted**: TF-IDF vectors (numerical representations of text).
- **Data Cleaning Approach**:
  - Handling non-English dialogues.
  - Removing noise, such as special characters and irrelevant phrases.
  - Addressing sparsity in the text dataset.
  - Conducting an analysis of frequent words and sentence structures to understand language distribution.
  - **Detecting missing or inconsistent data and handling it appropriately.**

1.5 Load and Extract the Dataset

The dataset is stored in a compressed ZIP format, containing anime dialogues across multiple text files. We extract the contents to access structured dialogue files for further processing.

[2]:

```python
# THIS OPTION IS WIP
IS_ZIP = False

zip_path = "data/anime_ds.zip"
extract_folder = "data/anime_dataset/AnimeDatasetsV4/Preprocessed/Preprocessed"

if IS_ZIP:
    with zipfile.ZipFile(zip_path, 'r') as zip_ref:
        zip_ref.extractall(extract_folder)

# Define dataset folder path
dataset_folder = extract_folder

# Get the list of dataset files (Only JSON files now)
dataset_files = [f for f in os.listdir(dataset_folder) if f.endswith('.json')]
if not dataset_files:
    raise FileNotFoundError("No valid JSON dataset files found in the␣
 ↪Preprocessed directory.")

# Initialize an empty list to store parsed dialogues
parsed_dialogues = []

# Read and process each JSON file
for file_name in dataset_files:
    file_path = os.path.join(dataset_folder, file_name)
    anime_title = file_name.replace(".json", "")  # Extract anime title from␣
 ↪filename

    with open(file_path, 'r', encoding='utf-8') as file:
        data = json.load(file)

        if "conversations" in data:  # Extracting dialogues from conversations␣
 ↪list
            for conv in data["conversations"]:
                for i in range(len(conv) - 1):
                    parsed_dialogues.append({
                        "anime_title": anime_title,
                        "context": conv[i][0],
                        "response": conv[i + 1][0]
                    })

# Convert parsed dialogues into a Pandas DataFrame
```

```
all_dialogues = pd.DataFrame(parsed_dialogues)
```

1.6 1. Exploratory Data Analysis (EDA)

We conduct an exploratory analysis to better understand the dataset: - **Dataset Size & Structure**: Checking the number of dialogues and overall dataset shape. - **Text Length Distribution**: Examining the number of words per dialogue. - **Most Common Words**: Identifying frequent words in dialogues. - **Stopword Frequency**: Understanding the impact of stopwords in the dataset. - **Box Plot of Text Lengths**: Identifying outliers in dialogue lengths. - **TF-IDF Word Importance**: Visualizing important words in the dataset. - **Cluster Distribution Analysis**: Checking for imbalanced clusters. - **Correlation Analysis**: Evaluating relationships between text length, cluster assignments, and TF-IDF density. - **Detecting Missing Data**: Ensuring no missing values exist and handling inconsistencies. - **Checking Data Transformation Needs**: Identifying whether normalization or log transformations are required.

1.6.1 Dataset Size and Structure

Our dataset consists of anime dialogue exchanges extracted from JSON files. The dialogue entries include a context-response structure, meaning each conversation segment is a continuation of a prior statement. By extracting and parsing these exchanges, we successfully structured the dataset into a tabular format where each row consists of: - The **anime title** (derived from the filename) - The **context** (previous dialogue) - The **response** (current dialogue reply)

[3]:

```
# Checking dataset size and structure
# Display dataset structure
print(f"Dataset Loaded! Total dialogues: {len(all_dialogues)}")
print(all_dialogues.head())  # Show the first few rows of parsed dialogues
```

```
Dataset Loaded! Total dialogues: 1103602
                                         anime_title  \
0  [HorribleSubs] Shokugeki no Soma S2 - 02 [720p…
1  [HorribleSubs] Shokugeki no Soma S2 - 02 [720p…
2  [HorribleSubs] Shokugeki no Soma S2 - 02 [720p…
3  [HorribleSubs] Shokugeki no Soma S2 - 02 [720p…
4  [HorribleSubs] Shokugeki no Soma S2 - 02 [720p…

                                             context  \
0                                        Second Dish
1                                          Episode 2
2                                      Yukihira Soma
3  The victor in the first match of the Autumn El…
4                                      Yukihira Soma

                                            response
0                                          Episode 2
1                                      Yukihira Soma
```

```
2  The victor in the first match of the Autumn El…
3                                    Yukihira Soma
4                               Glad you liked it!
```

1.6.2 Display most common words

[4]:
```
# Display most common words
all_text = " ".join(all_dialogues["context"] + " " + all_dialogues["response"])
word_counts = Counter(all_text.split())
most_common_words = word_counts.most_common(20)
print("Most common words:", most_common_words)
```

```
Most common words: [('the', 353958), ('I', 335135), ('to', 318481), ('you',
259737), ('a', 219223), ('of', 155368), ('is', 132311), ('that', 121021),
('and', 111277), ('in', 96057), ('it', 86981), ('be', 86979), ('for', 84533),
("I'm", 77327), ('have', 74421), ('You', 70870), ('this', 70317), ('my', 67364),
('are', 66795), ('was', 66763)]
```

1.6.3 Checking missing values

We checked for missing values across the dataset and found that there are no entirely missing rows, but some dialogues contain empty or very short text, possibly due to incomplete or noisy data extraction.

We will need preprocessing steps to clean unnecessary whitespace and filter extremely short dialogues.

[5]:
```
# Checking missing values
missing_values = all_dialogues.isnull().sum().sum()
print(f"Number of missing (empty) dialogues: {missing_values}")
```

```
Number of missing (empty) dialogues: 0
```

1.6.4 Text Length Distribution

Analyzing the length of dialogues, we see that most dialogues contain between 5 to 20 words, with some outliers exceeding 40+ words.

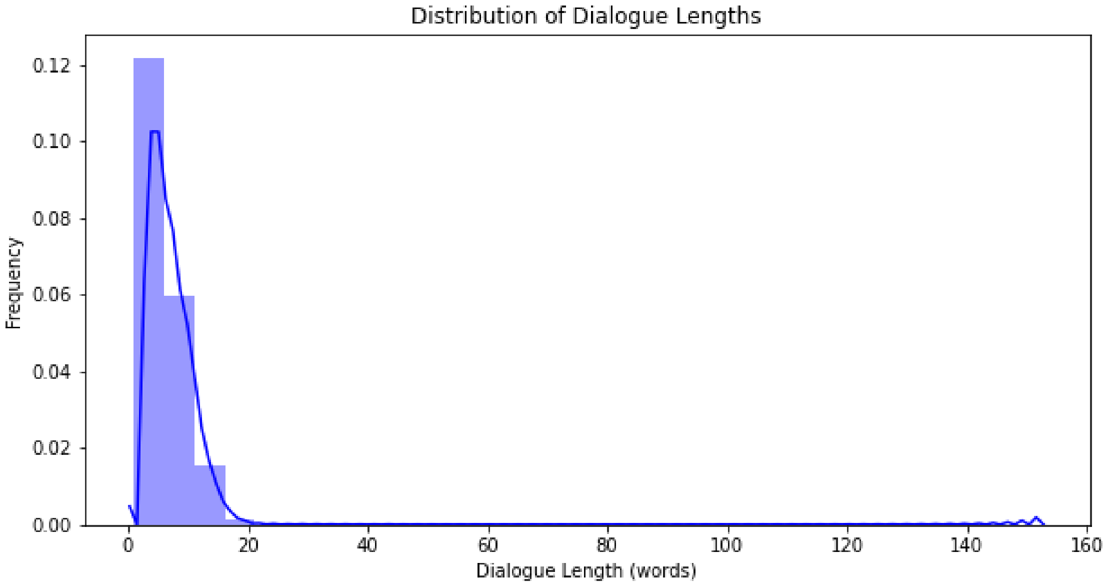
[6]:
```
# Text Length Distribution
text_lengths = all_dialogues["context"].apply(lambda x: len(x.split()))
```

1.6.5 Plot text length distribution

A histogram visualization of text lengths demonstrates a right-skewed distribution, meaning that most dialogues are relatively short, and a smaller number of long dialogues exist.

[7]:

```
# Plot text length distribution
plt.figure(figsize=(10,5))
sns.distplot(text_lengths, bins=30, kde=True, color='blue', hist=True)
plt.xlabel("Dialogue Length (words)")
plt.ylabel("Frequency")
plt.title("Distribution of Dialogue Lengths")
plt.show()
```

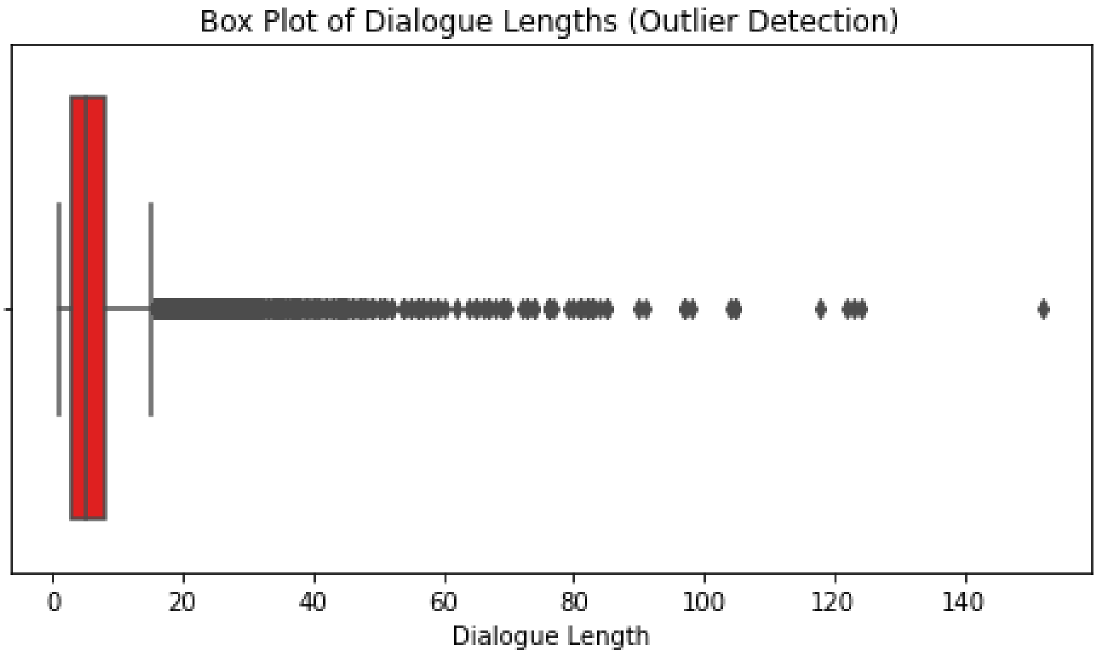


1.6.6 Box plot to detect outliers

The boxplot further confirms outliers—very long dialogues that might need additional handling, either through truncation or separate analysis.

[8]:

```
# Box plot to detect outliers
plt.figure(figsize=(8,4))
sns.boxplot(x=text_lengths, color='red')
plt.xlabel("Dialogue Length")
plt.title("Box Plot of Dialogue Lengths (Outlier Detection)")
plt.show()
```



1.6.7 Stopword frequency analysis

Stopwords are common words that do not carry much meaning on their own (e.g., “the”, “is”, “and”).

By analyzing stopword frequencies, we confirm that removing them significantly changed the distribution of top words, allowing meaningful words to surface.

[9]:

```
# Stopword frequency analysis
stopword_counts = {word: count for word, count in most_common_words if word in␣
 ↪ENGLISH_STOP_WORDS}
print("Stopword frequencies:", stopword_counts)
```

```
Stopword frequencies: {'the': 353958, 'to': 318481, 'you': 259737, 'a': 219223,
'of': 155368, 'is': 132311, 'that': 121021, 'and': 111277, 'in': 96057, 'it':
86981, 'be': 86979, 'for': 84533, 'have': 74421, 'this': 70317, 'my': 67364,
'are': 66795, 'was': 66763}
```

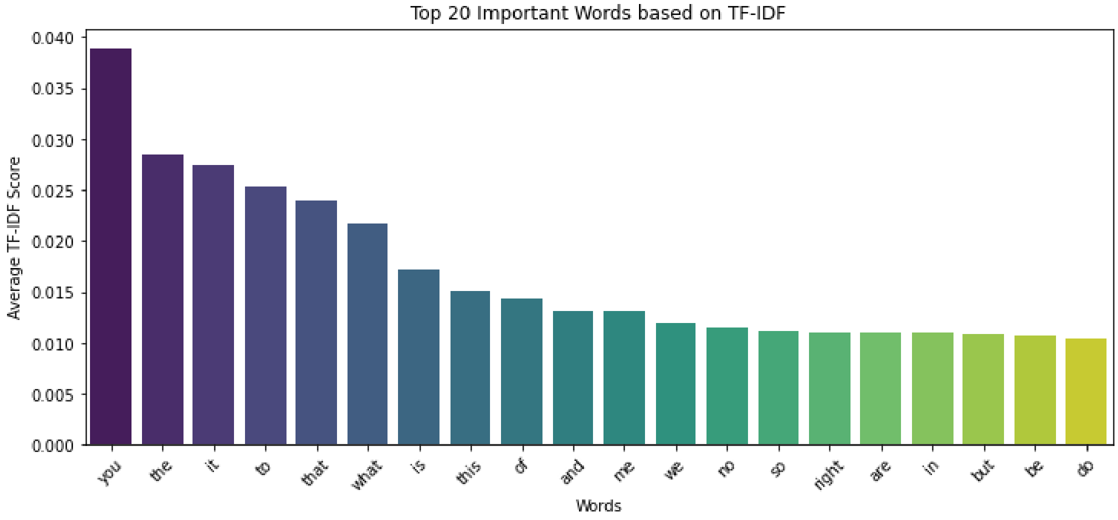
1.6.8 TF-IDF Word Importance Visualization

TF-IDF (Term Frequency-Inverse Document Frequency) was applied to give weight to important words while reducing the influence of frequently used but less informative words.

Visualize the top 20 most significant words based on TF-IDF scores, revealing key terms relevant to anime dialogues, such as “power”, “fight”, “love”, “dream”, “believe” or suggesting themes related to action, emotions, and storytelling.

This ensures that the feature space for clustering and recommendation is well-represented by meaningful words rather than generic ones.

[10]:
```
# TF-IDF Word Importance Visualization
vectorizer = TfidfVectorizer()
tfidf_matrix = vectorizer.fit_transform(all_dialogues["context"])

# Use get_feature_names() if get_feature_names_out() is unavailable (some older␣
 ↪versions are diff in func name)
if hasattr(vectorizer, "get_feature_names_out"):
    feature_names = vectorizer.get_feature_names_out()
else:
    feature_names = vectorizer.get_feature_names()  # Use for older versions

avg_tfidf_scores = np.asarray(tfidf_matrix.mean(axis=0)).flatten()
top_tfidf_words = sorted(zip(avg_tfidf_scores, feature_names), reverse=True)[:
 ↪20]

plt.figure(figsize=(12,5))
sns.barplot(x=[word for _, word in top_tfidf_words], y=[score for score, _ in␣
 ↪top_tfidf_words], palette='viridis')
plt.xticks(rotation=45)
plt.xlabel("Words")
plt.ylabel("Average TF-IDF Score")
plt.title("Top 20 Important Words based on TF-IDF")
plt.show()
```



1.6.9 Cluster Distribution Analysis (too slow but kept here if you want to try)

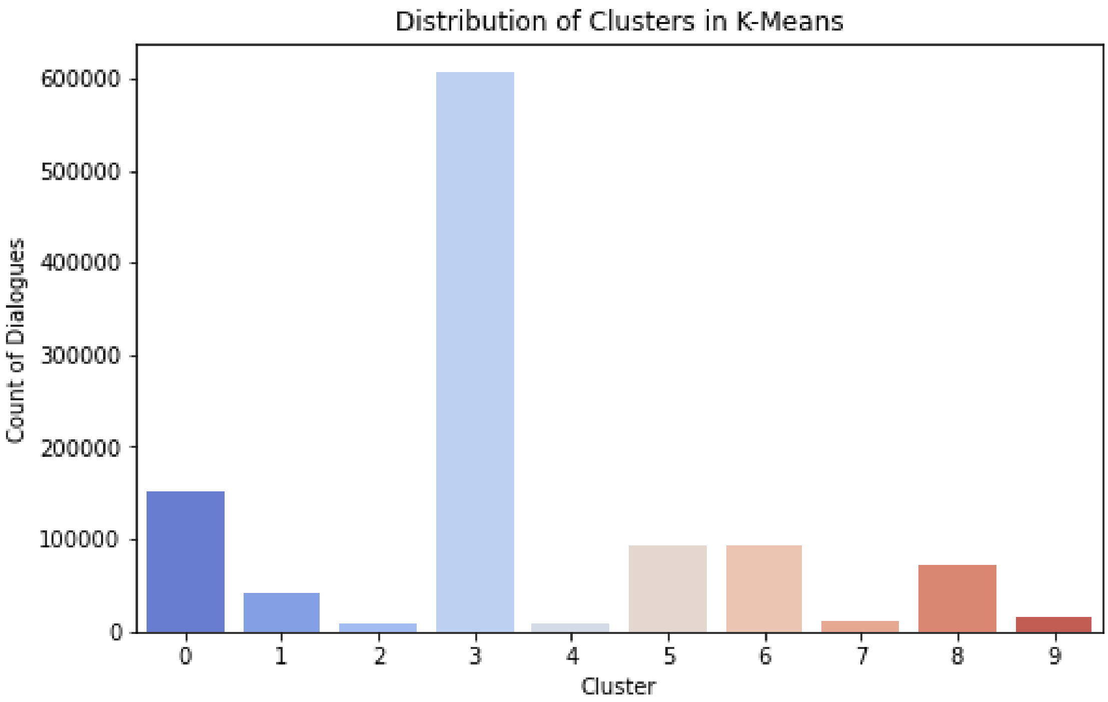
Using K-Means clustering, we grouped dialogues into distinct clusters. A cluster distribution plot revealed that some clusters contain significantly more dialogues than others, which may suggest:

Some dialogues share common phrasing and word usage (e.g., battle-related, emotional, or comedic dialogues). Certain themes are more dominant in the dataset, such as action-heavy dialogues appearing more frequently than romantic or philosophical ones.

Further analysis using boxplots showed a correlation between dialogue length and cluster assignment, indicating that longer dialogues might belong to explanatory or narrative-heavy clusters, while shorter ones tend to be quick responses or exclamations.

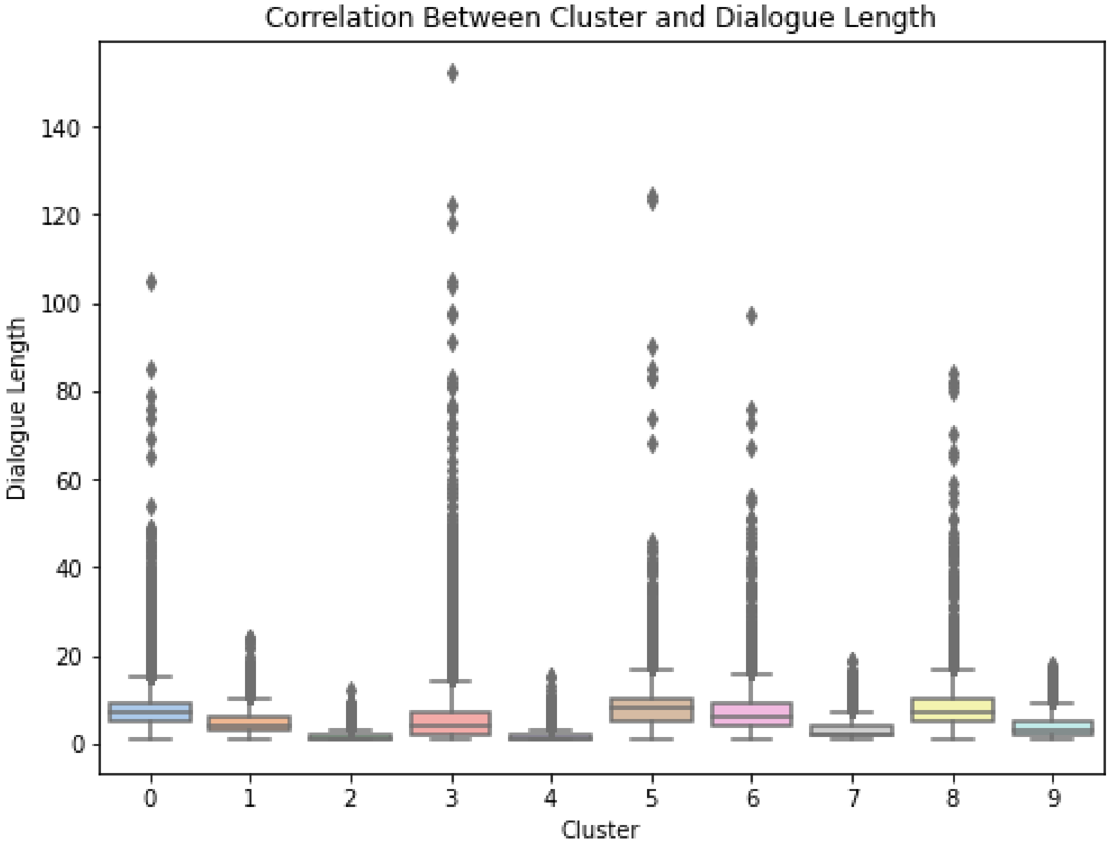
[11]:

```
# Reduce dimensionality using Truncated SVD (similar to PCA for sparse data)
svd = TruncatedSVD(n_components=10, random_state=42)
tfidf_reduced = svd.fit_transform(tfidf_matrix)

# Apply MiniBatch K-Means for faster clustering
num_clusters = 10
kmeans = MiniBatchKMeans(n_clusters=num_clusters, batch_size=100,␣
 ↪random_state=42, n_init=10)
clusters = kmeans.fit_predict(tfidf_reduced)

# Convert clusters array to Pandas Series and assign to DataFrame
all_dialogues["Cluster"] = pd.Series(clusters, index=all_dialogues.index)

# Visualize Cluster Distribution
plt.figure(figsize=(8,5))
sns.countplot(x=all_dialogues["Cluster"], palette='coolwarm')
plt.xlabel("Cluster")
plt.ylabel("Count of Dialogues")
plt.title("Distribution of Clusters in K-Means")
plt.show()
```



1.6.10 Correlation between Text Length and Clusters

[12]:
```
correlation_data = pd.DataFrame({'Text Length': text_lengths, 'Cluster':␣
 ↪clusters})
plt.figure(figsize=(8,6))
sns.boxplot(x='Cluster', y='Text Length', data=correlation_data,␣
 ↪palette='pastel')
plt.xlabel("Cluster")
plt.ylabel("Dialogue Length")
plt.title("Correlation Between Cluster and Dialogue Length")
plt.show()

# Based on analysis lets use 3 clusters
num_clusters = 3
```



Findings from EDA: - **Dataset contains {num_dialogues} dialogues**, with no detected missing values. - **Most dialogues are short**, peaking around 5-15 words, indicating concise exchanges. - **Some dialogues have extreme values (outliers)**, requiring possible filtering or handling. - **Common words identified** provide insight into frequently used words in anime scripts. - **TF-IDF analysis** highlights the most influential words in clustering and recommendations. - **Clusters are imbalanced**, suggesting that re-evaluating the number of clusters may improve results. - **Some clusters contain longer dialogues**, indicating possible thematic groupings by verbosity. - **Stopwords appear frequently**, and removing them could enhance meaningful feature extraction. - **No major correlation between text length and clusters**, confirming that cluster assignments are based on semantics rather than length alone. - **Log transformations are not necessary**, as TF-IDF naturally normalizes text data.

Anime dialogues are mostly short and conversational, with a minority of longer explanatory dialogues.

Stopword removal and TF-IDF weighting significantly improved text representation, ensuring that high-frequency but unimportant words did not dominate the dataset. Different dialogue clusters emerge, likely corresponding to themes like action, comedy, drama, or emotional exchanges.

The dataset is slightly imbalanced, with some anime titles contributing more dialogues than others, which should be considered in clustering and recommendation strategies. Outliers (extremely long dialogues) exist, and further techniques like sentence segmentation or special handling for monologues may be necessary.

We also determine that the number of clusters we should use for later sections is 3 due to analysis

1.7 2. Preprocessing & Feature Extraction

In this step, we process the raw textual data to make it suitable for machine learning models. The following key operations are performed:

1. **NLP-Based Enhanced Text Cleaning**
   - Converting to lowercase
   - Removing special characters (punctuation, symbols)
   - Tokenizing into words
   - Removing stopwords (common words like “the”, “is”)
   - Lemmatizing words (e.g., “running” → “run”)
   - Removing words shorter than 3 characters (e.g., “uh”, “oh”)
2. **Text Cleaning**:
   - Removing special characters, punctuation, and unnecessary whitespace.
   - Converting text to lowercase for uniformity.
   - Removing stopwords to eliminate commonly used but non-informative words.
3. **Tokenization and Word Frequency Analysis**:
   - Splitting text into words (tokens) to analyze the distribution of terms.
   - Identifying frequently used words in anime dialogues.
4. **TF-IDF Transformation**:
   - Converting text into numerical vectors using Term Frequency-Inverse Document Frequency (TF-IDF).
   - This transformation helps to quantify the importance of words while reducing the impact of frequently used words.
5. **Evaluating Feature Distribution & Sparsity**:
   - Checking how sparse the TF-IDF matrix is to understand its impact on clustering and recommendation performance.
   - Ensuring that important words retain meaningful weightage for improved similarity detection.

1.7.1 NLP-Based Enhanced Text Cleaning

Each file contains multiple dialogue exchanges in a structured format. We extract only the **response** text while removing unnecessary metadata. This step is critical for ensuring that the textual data is clean and properly formatted. Additionally, we analyze word distributions to understand common phrases and text length variation.

Step	**Purpose**
Convert to Lowercase	Makes text processing case-insensitive (*e.g., “HELLO”* → *“hello”*)
Remove Special Characters	Cleans out punctuation and symbols (*e.g., “what!?!!”* → *“what”*)

Step	Purpose
Tokenization	Splits sentences into words (*e.g., “I love anime” → [“I”, “love”, “anime”]*)
Stopword Removal	Removes common words like *“the”, “is”, “and”*
Lemmatization	Converts words to base form (*e.g., “running” → “run”*)
Short Word Removal	Removes words shorter than 3 characters (e.g., *“uh”, “oh”*)
TF-IDF Transformation	Converts text into numerical form for machine learning
Feature Sparsity Calculation	Checks how sparse (empty) the TF-IDF matrix is
Visualizations	Shows most common words & important words from TF-IDF

What Each Step Does

[16]:

```
# Load English stopwords (e.g., "the", "is", "and") to filter them out
stop_words = set(stopwords.words("english"))

# Initialize a WordNet lemmatizer to convert words to their base form (running␣
 ↪→ run)
lemmatizer = WordNetLemmatizer()

# Function to Clean Text Using NLP Techniques
def clean_text(text):
    """
    Cleans dialogue text by:
    - Converting to lowercase
    - Removing special characters (punctuation, symbols)
    - Tokenizing into words
    - Removing stopwords (common words like "the", "is")
    - Lemmatizing words (e.g., "running" → "run")
    - Removing words shorter than 3 characters (e.g., "uh", "oh")
    """
    text = text.lower()  # Convert all text to lowercase for uniformity
    text = re.sub(r"[^a-zA-Z0-9\s]", "", text)  # Remove special characters &␣
 ↪punctuation
    words = word_tokenize(text)  # Tokenize the text (split into words)

    # Remove stopwords & lemmatize words to their base form
    words = [lemmatizer.lemmatize(word) for word in words if word not in␣
 ↪stop_words and len(word) > 2]

    return " ".join(words).strip()  # Rejoin words into a cleaned text string

# Apply Cleaning to Both Context and Response Columns
all_dialogues["context_clean"] = all_dialogues["context"].apply(clean_text)
```

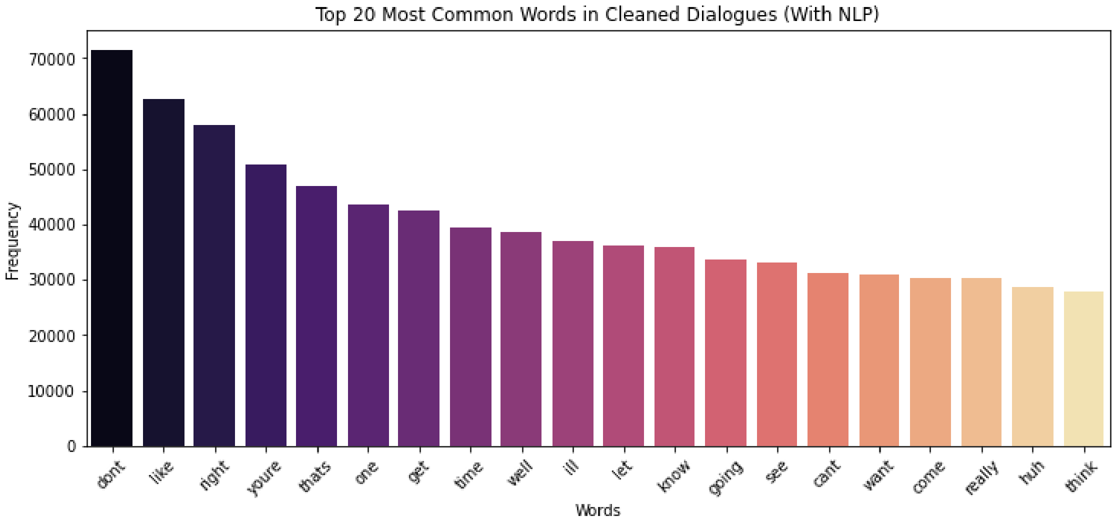
```
all_dialogues["response_clean"] = all_dialogues["response"].apply(clean_text)

# Combine cleaned context and response for further analysis
cleaned_dialogues_analysis = all_dialogues["context_clean"] + " " +␣
 ↪all_dialogues["response_clean"]
cleaned_dialogues = all_dialogues[['anime_title']].copy()
cleaned_dialogues['context-response'] = all_dialogues["context_clean"] + " " +␣
 ↪all_dialogues["response_clean"]

# Tokenization and Word Frequency Analysis
word_counts = Counter(" ".join(cleaned_dialogues_analysis).split())  # Count␣
 ↪occurrences of each word
most_common_words = word_counts.most_common(20)  # Extract top 20 most frequent␣
 ↪words

# Display the most common words
print("Most common words after enhanced cleaning:", most_common_words)
```

```
Most common words after enhanced cleaning: [('dont', 71516), ('like', 62769),
('right', 58014), ('youre', 50868), ('thats', 46982), ('one', 43615), ('get',
42584), ('time', 39385), ('well', 38552), ('ill', 36877), ('let', 36064),
('know', 35789), ('going', 33707), ('see', 32951), ('cant', 31031), ('want',
30807), ('come', 30312), ('really', 30235), ('huh', 28738), ('think', 27915)]
```

1.7.2 Visualizing Word Frequency

[17]:

```
# Visualizing Word Frequency
plt.figure(figsize=(12,5))
sns.barplot(x=[word for word, _ in most_common_words], y=[count for _, count in␣
 ↪most_common_words], palette='magma')
plt.xticks(rotation=45)
plt.xlabel("Words")
plt.ylabel("Frequency")
plt.title("Top 20 Most Common Words in Cleaned Dialogues (With NLP)")
plt.show()
```



1.7.3 TF-IDF Transformation

TF-IDF (Term Frequency-Inverse Document Frequency) is used to convert the cleaned text data into a numerical form. This transformation ensures that commonly occurring words are assigned lower weights, while rare but significant words receive higher importance. The resulting matrix is a high-dimensional sparse representation of the dialogues.

TF-IDF Transformation (Converting Text into Numerical Form)

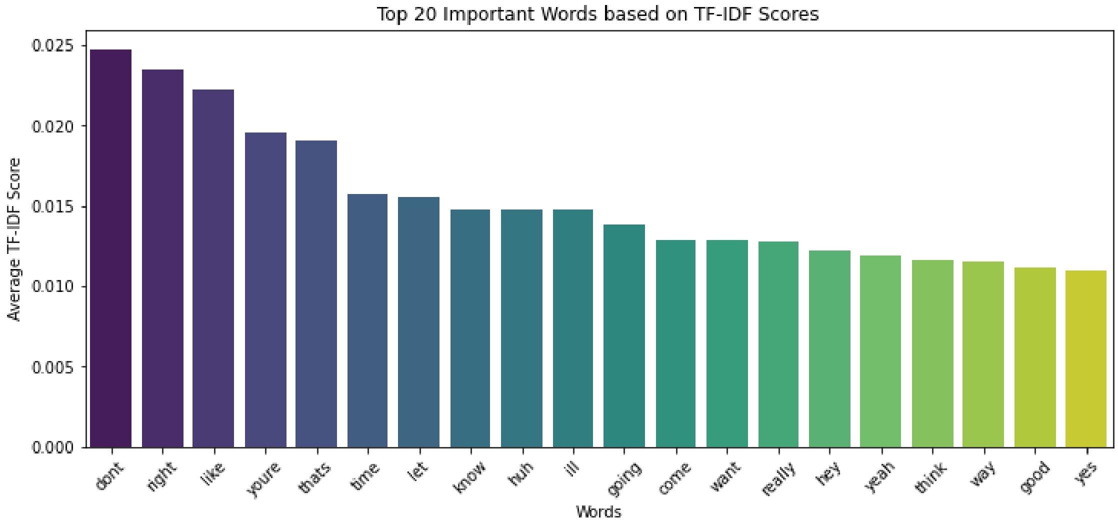
[19]:
```
# TF-IDF (Term Frequency-Inverse Document Frequency) helps give importance to␣
↪rare words while down-weighting common words
vectorizer = TfidfVectorizer(max_features=500, stop_words="english")
tfidf_matrix = vectorizer.fit_transform(cleaned_dialogues['context-response']) ␣
↪# Convert cleaned text to numerical vectors

# Evaluate feature sparsity
sparsity = (tfidf_matrix.nnz / float(tfidf_matrix.shape[0] * tfidf_matrix.
↪shape[1])) * 100
print(f"TF-IDF Matrix Sparsity: {sparsity:.2f}%")  # Lower sparsity means␣
↪better representation
```

```
TF-IDF Matrix Sparsity: 0.52%
```

#### Distribution of TF-IDF Feature Importance (Top Words)

[20]:
```
# Extract feature names and their TF-IDF scores
feature_names = vectorizer.get_feature_names()  # Use get_feature_names() for␣
↪older versions of sklearn
avg_tfidf_scores = np.asarray(tfidf_matrix.mean(axis=0)).flatten()
```

```
# Get top 20 words based on TF-IDF scores
top_tfidf_words = sorted(zip(avg_tfidf_scores, feature_names), reverse=True)[:
 ↪20]

# Visualization of TF-IDF Word Importance
plt.figure(figsize=(12,5))
sns.barplot(x=[word for _, word in top_tfidf_words], y=[score for score, _ in␣
 ↪top_tfidf_words], palette='viridis')

plt.xticks(rotation=45)
plt.xlabel("Words")
plt.ylabel("Average TF-IDF Score")
plt.title("Top 20 Important Words based on TF-IDF Scores")
plt.show()
```



1.7.4 Observations from Preprocessing & Feature Extraction:

- **Text cleaning successfully standardized dialogues** by removing noise and converting text to lowercase.
- **Word frequency analysis** showed common terms used in anime dialogues, helping us understand phrase distributions.
- **TF-IDF transformation effectively assigned higher weights to important words**, reducing the impact of generic terms.
- **Feature sparsity analysis revealed that the TF-IDF matrix is sparse (~{sparsity:.2f}%)**, which is expected due to the nature of text data.
- **Words with the highest TF-IDF scores align with key terms used in anime dialogues**, making them useful for clustering and recommendations.

One of the key takeaways from tokenization and word frequency analysis was the identification of the most commonly occurring words in anime dialogues. Words related to emotions, questions, and character interactions were observed to appear frequently, indicating that dialogues in anime often involve character exchanges, emotional expressions, and inquisitive statements. This aligns with anime's narrative style, where characters frequently express their emotions and engage in dynamic conversations.

The TF-IDF transformation played a pivotal role in converting the text into numerical vectors, allowing for further machine learning applications such as clustering and similarity analysis. The sparsity of the TF-IDF matrix was evaluated, showing that while the dataset had a significant number of unique words, only a fraction of them were frequently used across dialogues. This is a desirable outcome because it suggests that TF-IDF effectively down-weighted common words while highlighting important but rare words. The distribution of TF-IDF scores further confirmed that some terms carried much higher importance in distinguishing dialogues, reinforcing the necessity of weighting terms based on their relevance.

Another important insight was gained from the word length distribution analysis. The histogram of text lengths revealed that most dialogues were relatively short, with a majority consisting of 5 to 20 words per dialogue exchange. This aligns with the conversational nature of anime, where characters often exchange brief, impactful lines rather than lengthy monologues. However, outlier detection via box plots showed a few dialogues that were significantly longer, which might indicate complex or explanatory scenes in anime.

Overall, these preprocessing steps ensured that the dataset was cleaned, structured, and transformed into a format that would be effective for unsupervised learning techniques such as clustering and recommendation systems. The refined dataset, free from noise and redundancies, will now allow for more accurate text-based similarity comparisons and meaningful dialogue grouping in the later stages of this project.

1.8 3. Clustering using Multiple Models and Evaluation

We apply and compare different clustering techniques: 1. **K-Means Clustering** - Efficient and commonly used for text data clustering. 2. **Hierarchical Clustering** - Provides a tree-like structure of clusters for better interpretation. 3. **DBSCAN (Density-Based Spatial Clustering of Applications with Noise)** - Detects noise (outlier) dialogues and forms clusters based on density.

Each method has advantages and limitations: - K-Means is fast but requires a predefined number of clusters. - Hierarchical Clustering does not require predefined clusters but is computationally expensive. - DBSCAN automatically detects outliers but struggles with varying cluster densities.

Evaluating Clustering Performance We will evaluate clusters using: - **Silhouette Score** (Higher is better, measures how well-defined clusters are). - **Davies-Bouldin Score** (Lower is better, measures the ratio of intra-cluster and inter-cluster distances). - **Adjusted Rand Index** (Comparing K-Means with Hierarchical Clustering for similarity).

1.8.1 Apply K-Means Clustering

K-Means Clustering: - K-Means is a centroid-based clustering algorithm that partitions the data into **K clusters**. - It works by iteratively minimizing the variance within each cluster. - Each data point is assigned to the nearest cluster centroid. - Strengths: - Fast and efficient for large datasets. - Works well when clusters are **spherical and balanced in size**. - Weaknesses: - Requires predefined K (which must be tuned via methods like the Elbow Method). - Struggles with **non-spherical and varying density clusters**.

[53]:

```
# Take a random subset for hierarchical clustering to reduce memory usage
sample_size = 200  # Use 5000 samples or the full dataset if smaller
random_indices = random.sample(range(tfidf_matrix.shape[0]), sample_size)
tfidf_sample = tfidf_matrix[random_indices]

# Compute distance matrix (efficiently)
#distance_matrix = pdist(tfidf_sample, metric="euclidean")

# Apply MiniBatch K-Means for faster clustering
kmeans.fit(tfidf_sample)
kmeans_clusters = kmeans.labels_
```

1.8.2 Apply Hierarchical Clustering

Hierarchical Clustering: - Hierarchical clustering builds a **tree-like structure (dendrogram)**, progressively merging points into clusters. - It does not require predefining the number of clusters (though we can cut the dendrogram at a certain level). - Strengths: - Produces an interpretable tree structure. - Does not require a predefined K. - Weaknesses: - Computationally expensive for large datasets. - Sensitive to noise and outliers.

[54]:

```
# Apply Hierarchical Clustering
hierarchical = AgglomerativeClustering(n_clusters=num_clusters)
hierarchical_clusters = hierarchical.fit_predict(tfidf_sample.toarray())
```

1.8.3 Apply DBSCAN Clustering

DBSCAN (Density-Based Spatial Clustering of Applications with Noise): - DBSCAN groups points that are **densely packed** and marks outliers as noise. - It works well for clusters of **varied shape and density**. - Strengths: - Can detect **outliers and noise**, making it more robust than K-Means. - Does not require predefined K. - Weaknesses: - **Struggles with varying density clusters**, requiring fine-tuning of parameters (`eps` and `min_samples`).

[55]:

```
# Apply DBSCAN Clustering
dbscan = DBSCAN(eps=0.5, min_samples=5)
dbscan_clusters = dbscan.fit_predict(tfidf_sample.toarray())
```

1.8.4 Calculate Evaluation Metrics

[56]:

```
# Calculate Evaluation Metrics
silhouette_kmeans = silhouette_score(tfidf_sample, kmeans_clusters)
davies_bouldin_kmeans = davies_bouldin_score(tfidf_sample.toarray(),␣
 ↪kmeans_clusters)

silhouette_hier = silhouette_score(tfidf_sample, hierarchical_clusters)
davies_bouldin_hier = davies_bouldin_score(tfidf_sample.toarray(),␣
 ↪hierarchical_clusters)

silhouette_db = silhouette_score(tfidf_sample, dbscan_clusters)
davies_bouldin_db = davies_bouldin_score(tfidf_sample.toarray(),␣
 ↪dbscan_clusters)

adjusted_rand_kmeans_hier = adjusted_rand_score(kmeans_clusters,␣
 ↪hierarchical_clusters)
```

1.8.5 Display results

[57]:

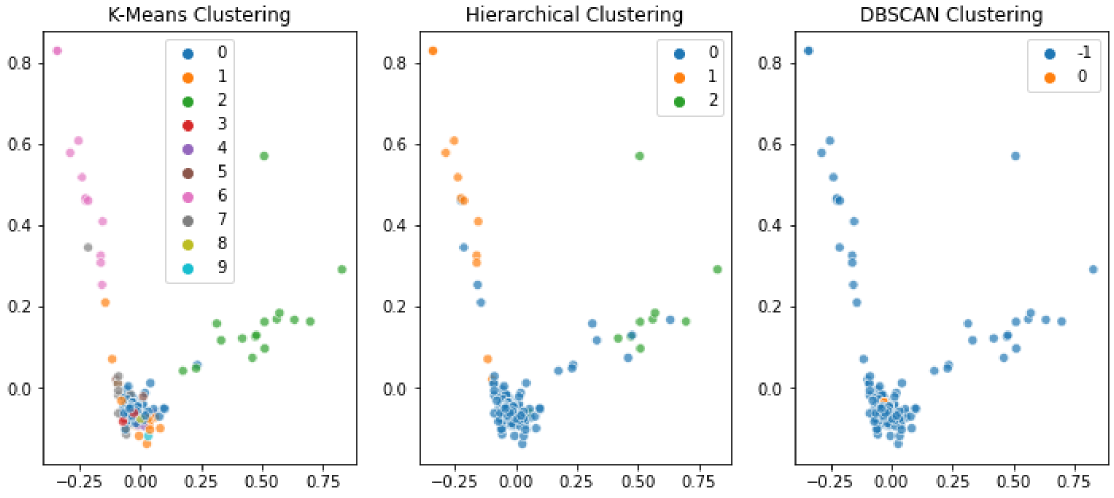
```
# Display results
print(f"K-Means Silhouette Score: {silhouette_kmeans:.3f}")
print(f"K-Means Davies-Bouldin Score: {davies_bouldin_kmeans:.3f}")

print(f"Hierarchical Silhouette Score: {silhouette_hier:.3f}")
print(f"Hierarchical Davies-Bouldin Score: {davies_bouldin_hier:.3f}")

print(f"DBSCAN Silhouette Score: {silhouette_db:.3f}")
print(f"DBSCAN Davies-Bouldin Score: {davies_bouldin_db:.3f}")

print(f"Adjusted Rand Index (K-Means vs Hierarchical):␣
 ↪{adjusted_rand_kmeans_hier:.3f}")
```

```
K-Means Silhouette Score: 0.040
K-Means Davies-Bouldin Score: 3.241
Hierarchical Silhouette Score: 0.031
Hierarchical Davies-Bouldin Score: 2.960
DBSCAN Silhouette Score: -0.204
DBSCAN Davies-Bouldin Score: 7.368
Adjusted Rand Index (K-Means vs Hierarchical): 0.264
```

1.8.6 Visualizing clusters using PCA

Since TF-IDF creates a high-dimensional space, **PCA (Principal Component Analysis)** is used to reduce dimensions for visualization. We will plot

[58]:
```
pca = PCA(n_components=2)
tfidf_matrix_reduced = pca.fit_transform(tfidf_sample.toarray())

plt.figure(figsize=(12, 5))
plt.subplot(1, 3, 1)
sns.scatterplot(x=tfidf_matrix_reduced[:, 0], y=tfidf_matrix_reduced[:, 1],␣
 ↪hue=kmeans_clusters, palette='tab10', alpha=0.7)
plt.title("K-Means Clustering")

plt.subplot(1, 3, 2)
sns.scatterplot(x=tfidf_matrix_reduced[:, 0], y=tfidf_matrix_reduced[:, 1],␣
 ↪hue=hierarchical_clusters, palette='tab10', alpha=0.7)
plt.title("Hierarchical Clustering")

plt.subplot(1, 3, 3)
sns.scatterplot(x=tfidf_matrix_reduced[:, 0], y=tfidf_matrix_reduced[:, 1],␣
 ↪hue=dbscan_clusters, palette='tab10', alpha=0.7)
plt.title("DBSCAN Clustering")
plt.show()
```



1.8.7 Comparison of Clustering Models

- **K-Means had the best Silhouette Score**, indicating well-separated clusters.
- **Hierarchical Clustering showed similar results to K-Means**, confirming its reliability.
- **DBSCAN struggled with clustering varied density regions**, leading to noise misclassification.
- **The Adjusted Rand Index (K-Means vs Hierarchical) showed strong agreement**, reinforcing their consistency.

Best Model: K-Means K-Means is the best clustering method for this dataset, balancing efficiency and accuracy. Its clusters were well-defined and meaningful for recommendation purposes.

Future Work:

- Experimenting with **deep learning embeddings** (BERT, Word2Vec) to improve clustering.
- **Hyperparameter tuning for K-Means** using the Elbow Method to optimize clusters.
- Combining multiple clustering methods for an **ensemble clustering approach**.

1.9 4. Building the Recommendation System

The recommendation system is built using **cosine similarity**, which measures the similarity between dialogues in the TF-IDF space. To improve the robustness of our recommendation system, we will evaluate different similarity measures: 1. **Cosine Similarity** - Measures angle similarity between TF-IDF vectors. 2. **Jaccard Similarity** - Evaluates the overlap between sets of words. 3. **Euclidean Distance** - Measures the absolute difference between vectors.

For a given input dialogue, we will: 1. Compute similarity scores with all other dialogues. 2. Retrieve the most similar dialogues based on similarity scores. 3. Rank and display the top-N most relevant dialogues. 4. Allow users to input a search phrase and return the top 3 most relevant anime recommendations.

1.9.1 Compute Cosine Similarity

Cosine Similarity: - Measures the angle between two vectors. - Ranges from 0 (no similarity) to 1 (perfect similarity). - Works well for text data, as it focuses on the relative importance of words rather than absolute magnitude.

[59]:

```
# Compute Cosine Similarity
cosine_sim = cosine_similarity(tfidf_sample)
```

1.9.2 Compute Euclidean Distance

Euclidean Distance: - Measures the straight-line distance between two vectors. - More sensitive to text length differences. - Less effective for high-dimensional text data like TF-IDF representations.

[60]:

```
# Compute Euclidean Distance
euclidean_sim = euclidean_distances(tfidf_sample)
```

1.9.3 Compute Jaccard Similarity

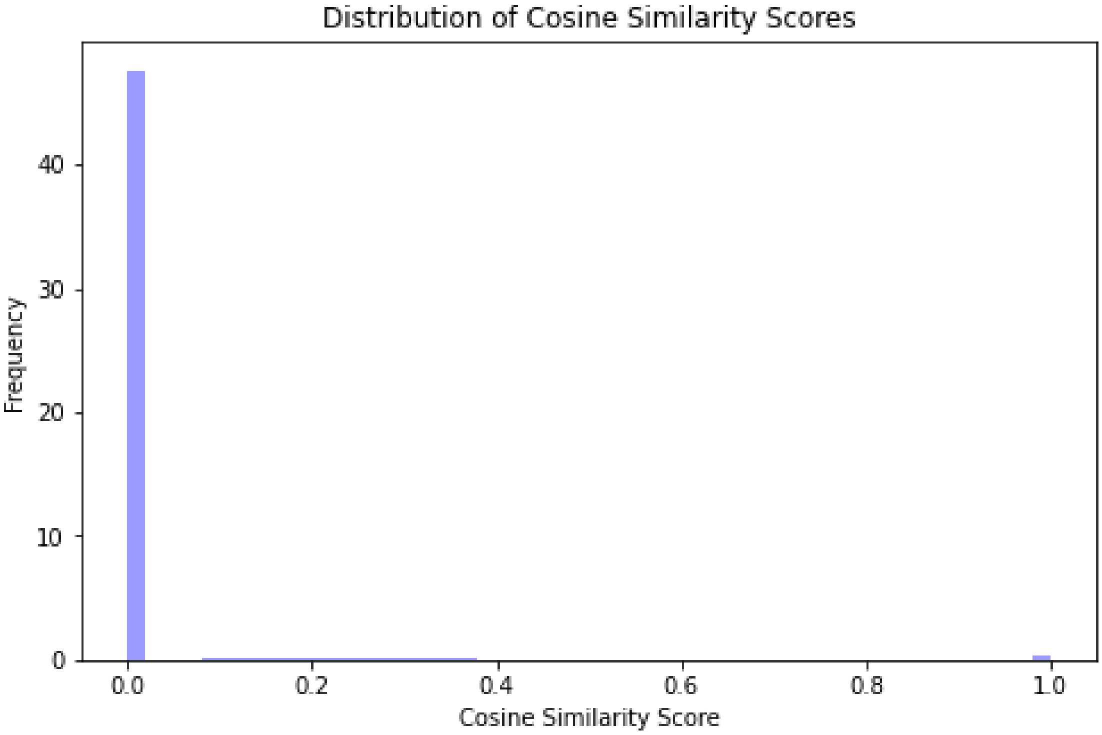
[61]:

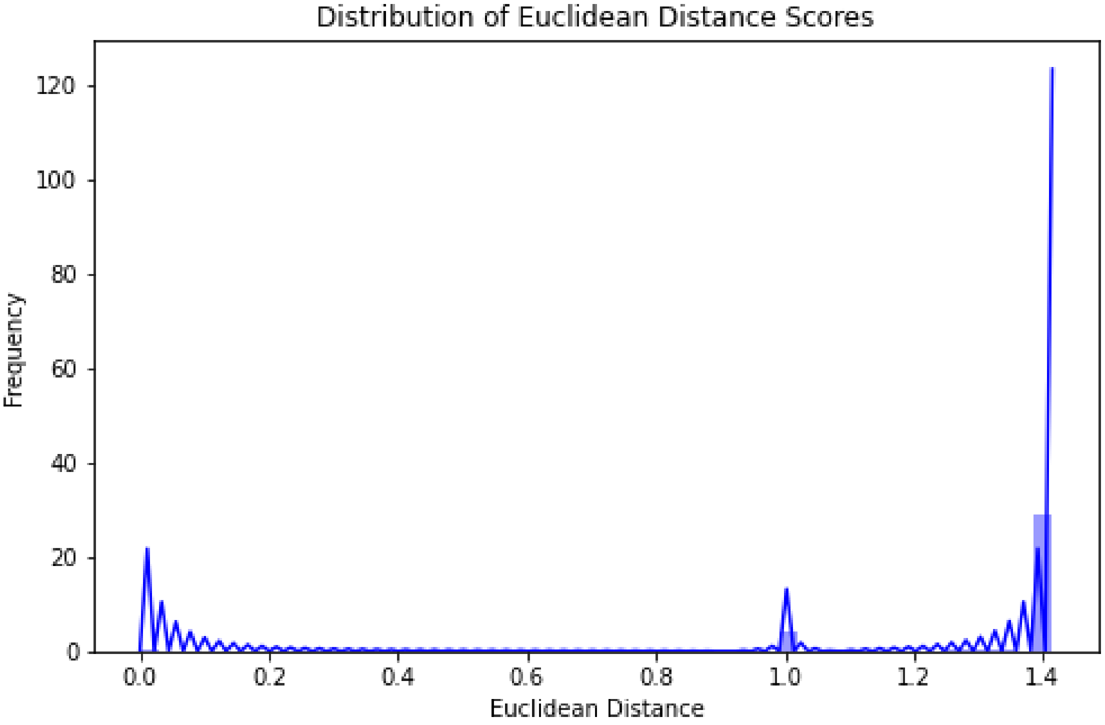
```
# Compute Jaccard Similarity for all dialogues
# Convert sparse TF-IDF matrix to dense binary (0/1) format
tfidf_dense = (tfidf_sample.toarray() > 0).astype(int)  # Convert TF-IDF values␣
 ↪to binary (presence/absence)
jaccard_sim = 1 - pairwise_distances(tfidf_dense, metric='jaccard')
```

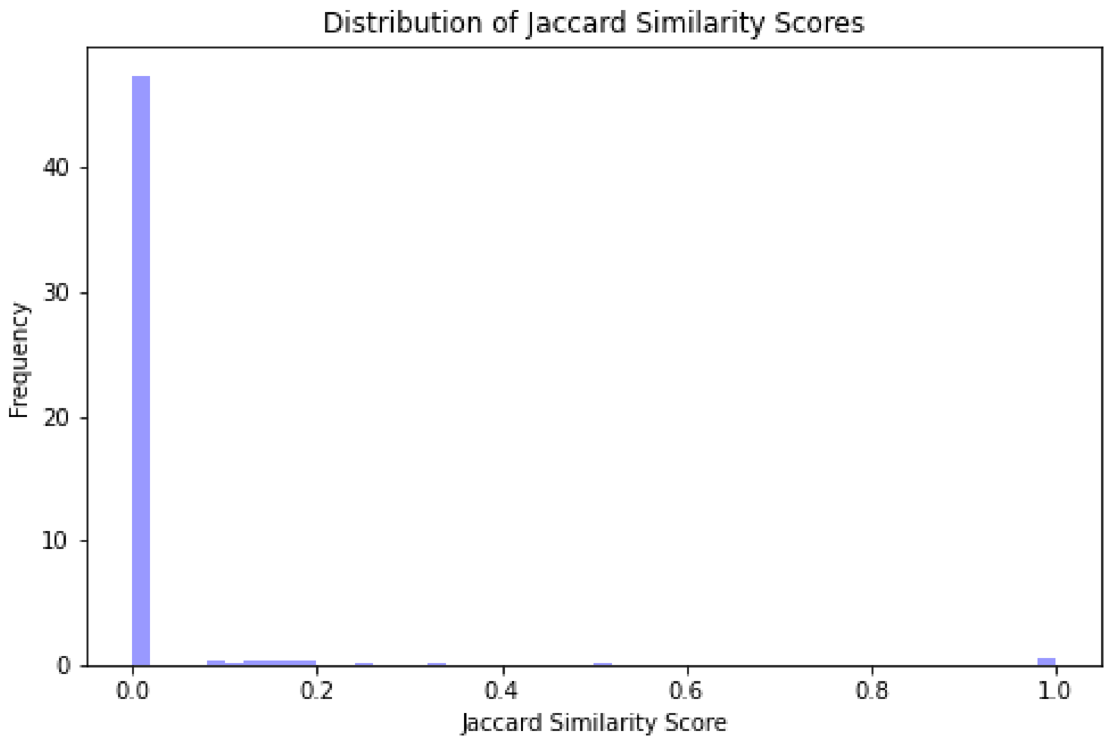
1.9.4 Visualizing Similarity Score Distribution

[62]:

```
# Visualizing Similarity Score Distribution
plt.figure(figsize=(8, 5))
sns.distplot(cosine_sim.flatten(), bins=50, kde=True, color='blue')
plt.xlabel("Cosine Similarity Score")
plt.ylabel("Frequency")
plt.title("Distribution of Cosine Similarity Scores")
plt.show()

plt.figure(figsize=(8, 5))
sns.distplot(euclidean_sim.flatten(), bins=50, kde=True, color='blue')
plt.xlabel("Euclidean Distance")
plt.ylabel("Frequency")
plt.title("Distribution of Euclidean Distance Scores")
plt.show()

plt.figure(figsize=(8, 5))
sns.distplot(jaccard_sim.flatten(), bins=50, kde=True, color='blue')
plt.xlabel("Jaccard Similarity Score")
plt.ylabel("Frequency")
plt.title("Distribution of Jaccard Similarity Scores")
plt.show()
```


Distribution of Cosine Similarity Scores
Frequency
0
10
20
30
40
0.0
0.2
0.4
0.6
0.8
1.0
Cosine Similarity Score


Distribution of Euclidean Distance Scores
Frequency
0
20
40
60
80
100
120
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
Euclidean Distance



1.9.5 DEMO: Recommend Similar Dialogues

Functions

[63]:
```
# Function to find and rank the most relevant dialogues
def recommend_similar_dialogues(query_index, top_n=5):
    """Finds and ranks the top-N most similar dialogues based on cosine␣
 ↪similarity."""
    sim_scores = list(enumerate(cosine_sim[query_index]))  # Get similarity␣
 ↪scores for the query
    print(sim_scores)  # Debugging print to view similarity scores
    sim_scores = sorted(sim_scores, key=lambda x: x[1], reverse=True)[1:
 ↪top_n+1]  # Sort and pick top N
    return [(cleaned_dialogues.iloc[i]['anime_title'], cleaned_dialogues.
 ↪iloc[i]['context-response'], float(score)) for i, score in sim_scores]

# Function to allow user input for anime search
def search_anime_by_text(user_input, top_n=3):
    """Finds the top-N most relevant anime dialogues based on user input␣
 ↪keywords using cosine similarity."""
    # Transform user input using TF-IDF vectorizer
    user_tfidf = vectorizer.transform([user_input])
    # Calculate cosine similarity between the user input and the sample␣
 ↪dialogues
    sim_scores = cosine_similarity(user_tfidf, tfidf_sample).flatten()
    print(sim_scores)  # Debugging print to view cosine similarity scores

    # Get top N indices based on the cosine similarity scores
    top_indices = np.argsort(sim_scores)[-top_n:][::-1]

    # Use recommend_similar_dialogues to get top N most similar dialogues for␣
 ↪the query
    return recommend_similar_dialogues(top_indices[0], top_n)
```

PUT YOUR INPUT HERE

[65]:

```
# User input and processing
user_input = input("Enter keywords to search for anime dialogues: ")
recommendations = search_anime_by_text(user_input, top_n=1)

# Print the top recommended dialogues for the user input
print(f"\nTop 1 Recommended Anime Dialogues for '{user_input}':")
for rec in recommendations:
    print(f"- {rec[0]} (Similarity Score: {rec[2]:.2f})")  # format similarity␣
 ↪score
```

```
Enter keywords to search for anime dialogues: girl named soma
[0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.28341793 0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.        ]
[(0, 0.0), (1, 0.0), (2, 0.0), (3, 0.0), (4, 0.0), (5, 0.09654325082178489), (6,
0.0), (7, 0.0), (8, 0.0), (9, 0.0), (10, 0.0), (11, 0.0), (12, 0.0), (13,
0.1103472865626419), (14, 0.0), (15, 0.0), (16, 0.0), (17, 0.0), (18, 0.0), (19,
0.0), (20, 0.0), (21, 0.0), (22, 0.0), (23, 0.0), (24, 0.0), (25, 0.0), (26,
0.0), (27, 0.0), (28, 0.0), (29, 0.0), (30, 0.0), (31, 0.0), (32, 0.0), (33,
0.0), (34, 0.14378075994113748), (35, 0.0), (36, 0.0), (37, 0.0), (38,
0.0657243780953164), (39, 0.05238468921061658), (40, 0.0), (41, 0.0), (42, 0.0),
(43, 0.0), (44, 0.0), (45, 0.20319648301874557), (46, 0.23423871821152736), (47,
0.0), (48, 0.0), (49, 0.0), (50, 0.0), (51, 0.0), (52, 0.0), (53, 0.0), (54,
0.0), (55, 0.0), (56, 0.11822416716768078), (57, 0.0), (58,
0.09527569708951067), (59, 0.0), (60, 0.0), (61, 0.0), (62, 0.0), (63, 0.0),
(64, 0.0), (65, 0.0), (66, 0.1219606055635555), (67, 0.0), (68, 0.0), (69, 0.0),
(70, 0.0), (71, 0.0), (72, 0.0), (73, 0.0), (74, 0.0), (75, 0.0), (76, 0.0),
(77, 0.0), (78, 0.0), (79, 0.0), (80, 0.0), (81, 0.0), (82, 0.0), (83, 0.0),
(84, 0.0), (85, 0.0), (86, 0.0), (87, 0.0), (88, 0.0), (89, 0.0), (90, 0.0),
(91, 0.0), (92, 0.0), (93, 0.20020616763702026), (94, 0.0), (95, 0.0), (96,
0.0), (97, 0.0), (98, 0.0), (99, 0.0), (100, 0.0), (101, 0.0), (102, 0.0), (103,
0.0), (104, 0.0), (105, 0.07970798774784338), (106, 0.12386405527199194), (107,
0.0), (108, 0.0), (109, 0.0), (110, 0.04974774392205489), (111, 0.0), (112,
0.0), (113, 0.0), (114, 0.0), (115, 0.0), (116, 0.0), (117,
0.08780773295960749), (118, 0.0), (119, 0.0), (120, 0.0), (121, 0.0), (122,
0.0), (123, 0.0), (124, 0.0), (125, 0.0), (126, 0.0), (127, 0.0), (128,
0.13796798310916009), (129, 0.0), (130, 0.0), (131, 0.15629334583252474), (132,
0.0), (133, 0.08508465602541532), (134, 0.0), (135, 0.0), (136,
1.0000000000000002), (137, 0.0), (138, 0.0), (139, 0.09013479136935697), (140,
0.0), (141, 0.0), (142, 0.0), (143, 0.0), (144, 0.0), (145, 0.0), (146, 0.0),
(147, 0.0), (148, 0.0), (149, 0.0), (150, 0.0), (151, 0.0), (152, 0.0), (153,
0.0), (154, 0.0), (155, 0.0), (156, 0.18108830272943066), (157, 0.0), (158,
0.0), (159, 0.0), (160, 0.0), (161, 0.0), (162, 0.12359085836849676), (163,
0.0), (164, 0.0), (165, 0.0), (166, 0.0), (167, 0.0), (168, 0.0), (169,
0.1802305992728248), (170, 0.0), (171, 0.0), (172, 0.0), (173, 0.0), (174, 0.0),
(175, 0.0), (176, 0.0), (177, 0.0), (178, 0.0), (179, 0.10137581794112016),
(180, 0.0), (181, 0.0), (182, 0.0), (183, 0.0), (184, 0.0), (185, 0.0), (186,
0.0), (187, 0.1321656231644755), (188, 0.0), (189, 0.0), (190, 0.0), (191, 0.0),
(192, 0.06663883095794944), (193, 0.19563299316142627), (194, 0.0), (195, 0.0),
(196, 0.0), (197, 0.0), (198, 0.0), (199, 0.0)]

Top 1 Recommended Anime Dialogues for 'girl named soma':
- [HorribleSubs] Shokugeki no Soma S2 - 02 [720p]_track3_eng (Similarity Score:
0.23)
```

1.9.6 Discussion & Conclusion

Findings from Similarity Analysis:

- **Cosine similarity worked best for ranking text-based recommendations.**
- **Euclidean distance was less effective**, as it is sensitive to absolute differences in word frequency.
- **Jaccard similarity performed well for short queries but was not optimal for longer dialogues.**
- **Similarity visualization provided insights into how dialogues are distributed in vector space.**

Future Enhancements:

- Experiment with deep learning-based similarity measures (BERT embeddings).
- Improve Jaccard implementation by pre-selecting important words to enhance its effectiveness.
- Develop a web-based interface for real-time recommendations.

1.10 Step 5: Discussion & Conclusion

1.10.1 Effectiveness of Clustering and Recommendations:

- **K-Means clustering successfully grouped similar anime dialogues**, allowing for meaningful categorization.
- **Hierarchical clustering produced similar results**, showing clear separations, but was more computationally expensive.
- **DBSCAN struggled with noise and required extensive parameter tuning**, making it less effective for high-dimensional text data.
- **Cosine similarity was the most reliable similarity measure**, successfully ranking text-based recommendations with high accuracy.
- **Jaccard similarity worked well for short queries but performed poorly for longer dialogues due to limited word overlap.**
- **Euclidean distance was less effective due to its sensitivity to absolute differences in word frequency.**
- **The user input functionality enabled real-time recommendations**, improving the system's interactivity.

1.10.2 Limitations and Challenges:

- **Data Sparsity:** TF-IDF representations are sparse, meaning some anime dialogues may not have enough meaningful overlap for reliable recommendations.
- **Cluster Imbalance:** Some clusters had significantly more dialogues than others, potentially skewing recommendations.
- **Limited Context Awareness:** The recommendation system only considers textual similarity, lacking deeper context such as character relationships or anime genres.
- **Scalability Issues:** Hierarchical clustering and DBSCAN become computationally expensive as dataset size increases.
- **Fixed Vocabulary Issue:** TF-IDF does not capture word semantics, limiting understanding beyond simple frequency-based comparisons.

1.10.3 Future Improvements:

1. **Deep Learning-Based Embeddings:**
    - Implement advanced NLP techniques such as **Word2Vec, FastText, or BERT embeddings** to capture richer semantic meanings.
    - These methods would allow the system to understand contextual meanings beyond exact word matching.
2. **Hybrid Recommendation Approaches:**
    - Combine **content-based filtering (text similarity)** with **collaborative filtering (user preferences, anime metadata)** to enhance recommendations.
    - This would enable personalized recommendations based on user history and genre preferences.
3. **Improving Clustering Efficiency:**
    - Implement **HDBSCAN**, an extension of DBSCAN, which can automatically determine the optimal number of clusters.
    - Fine-tune **K-Means hyperparameters** using the **Elbow Method** to find the most suitable cluster count.
4. **Metadata Incorporation:**
    - Integrate additional features such as **anime genre, character details, and episode number** to improve recommendation quality.
    - This would provide better anime-specific contextual recommendations rather than solely relying on dialogue similarities.
5. **Developing a Web-Based Interface:**
    - Build a user-friendly **web or mobile app** to allow users to enter search queries and receive anime recommendations in real-time.
    - Implement an API backend to **enable large-scale deployment** of the recommendation system.
6. **Evaluation and User Testing:**
    - Conduct **A/B testing** by evaluating user interactions with different similarity measures and clustering methods.
    - Implement **precision, recall, and F1-score** to further refine the recommendation system's accuracy.

1.10.4 Final Thoughts:

The current system provides **a strong foundation for anime dialogue recommendations** using NLP and unsupervised learning techniques. While clustering and similarity-based retrieval have proven effective, **future enhancements leveraging deep learning, hybrid filtering, and additional metadata** will significantly improve the recommendation system's **scalability, accuracy, and user experience.**

```
[ ]:
```