EAST WEST UNIVERSITY

Lab 2

Data Cleaning & Preprocessing for YouTube Text Mining

Course Title: *Data Mining*

Course Code: *CSE 477*

Semester: *Fall 2025*

Section: *01*

Submitted by

Name: *Nushrat Jaben Aurnima*

ID: *2022-2-60-146*

Date: *24/10/2025*

Submitted to

Amit Mandal

Lecturer, Department of Computer Science and Engineering

East West University

Table of Contents

Introduction 2

Objective 2

Tools and Libraries 2

Dataset Description 3

Methodology 3

- Profiling the Data 3
- Parsing Comments and Captions 4
- Cleaning and Preprocessing Pipeline 4
- Exporting Cleaned Data 4

Code Implementation 5

- Parsing Comments 5
- Parsing Captions 6
- Cleaning Functions 7

Results and Discussion 10

- Dataset Overview 10
- Tokenization Examples 10
- Stopword Removal 11
- Frequent Tokens 11
- Data Quality Check 11
- Engagement Analysis 11
- Most-Liked Comments 12
- Challenges Faced and Solutions 12

Conclusion 12

Introduction

In this week's lab ***"Data Cleaning & Preprocessing for YouTube Text Mining",*** we continued to work on the dataset created using the selected YouTube video from the previous lab ***"Exploratory Data Analysis on YouTube Data"***. Although in this lab, we tried a different approach for data cleaning using ***Natural Language Toolkit***, a Python library used for text processing tasks (tokenization, stemming, lemmatization) and parsing. The difference is that the previous cleaning was done manually with the help of basic Python functions such as string translation, regular expressions, and loops. I worked with raw YouTube comments and captions, which contained noisy elements like usernames, timestamps, emojis, and repeated words. The goal was to learn how to convert this unstructured data into a clean and organized format suitable for later data mining tasks. The goal of this lab was to learn efficient text cleaning designed for natural language processing.

Objective

- Comparing basic manual cleaning techniques with advanced natural language processing tools.
- Cleaning and organizing the cleaned comments and captions into structured DataFrames.
- Improving data quality through normalization, tokenization, stopword removal, and lemmatization.
- Handling inconsistencies such as emojis, extra spaces etc.
- Exporting cleaned datasets for text mining and pattern analysis tasks.

Tools and Libraries

- **Python 3.10** for running all the code.
- Installed libraries using `pip`:
    - **pandas** – creating DataFrames, and many more.
    - **matplotlib** – basic visualizations.
    - **nltk** – text processing tasks.
    - **webvtt-py** – reading and parsing. vtt caption files from YouTube videos.
- Additional modules:

    - **re** – cleaning text using regular expressions.
    - **datetime (timedelta)** – managing timestamps.
    - **collections.Counter** – counting tokens and word frequencies.
    - **nltk.bigrams** – experimenting with basic word pair generation.
- NLTK data such as **punkt**, **stopwords**, **wordnet**, **omw-1.4**, and **punkt_tab** were downloaded.
- The whole setup was done on **Google Colab**.

Dataset Description

This created a file named **captions.en.vtt** that contains all the subtitles from the video. After that I used the **YouTube Data API v3** with the **google-api-python-client** library to collect comments. The script helped to fetch each comment along real time data such as timestamp and number of likes on comments.

The **captions.en.vtt** file contains the subtitles extracted from the video. The subtitles were mostly in a single continuous line, making sentence framing difficult. Some lines were merged while others had missing or blank lines between timestamps. Also had escape characters html command breaks. This required additional preprocessing to split and organize the captions properly so that each sentence read as a separate record.

The **comments.txt** file holds entry includes commenter's username, when the comment was posted, and the actual comment text. During inspection, I found some noticeable anomalies such as extra blank lines, and inconsistent spacing. A few comments also had special symbols and emojis that needed to be removed before analysis.

Methodology

Profiling the Data

First, I opened both files `all_comments.txt` and `captions.en.vtt` to get the idea of the files structure. In one hand, the comments file contained usernames, timestamps, and comment text, but also had blank lines and all were in one-line compromising readability. Similarly, the captions file was mostly in one long line with missing breaks and inconsistent spacing. Profiling helpe to identify the required fixes before parsing them into data frames.

Parsing Comments and Captions

After primary inspection, the raw comments were converted into structured **DataFrames**. It had attributes like username time stamp, word count, starting and ending time, duration and so on. block. For the captions, I used the **webvtt-py** library to extract the subtitle text. Since some lines were merged or blank, a fallback method was added to read the file manually and split it into proper sentences. Both DataFrames were then checked to confirm the text loaded correctly.

Cleaning and Preprocessing Pipeline

- **Preprocessing setup**: Using NLKT import libraries I defined some common English words as stops words. defining common English stopwords. Then prepared text-simplification tools to reduce words to their base form.
- **Text normalization**: Then each sentence (both comment and caption) is converted to lowercase, symbols and punctuation are removed, extra spaces are collapsed. This creates a uniform text structure before tokenization.
- **Tokenization**: After normalized text is broken into individual words known as okens. Example, *"This course is excellent."* → ["this", "course", "is", "excellent"].
- **Stopword removal**: Common filler words like in English language are filtered out, to limit only meaningful tokens which could contribute in the overall context.
- **Lemmatization**: This process converts to its base form (running → run) or stems to its root (learning → learn). This ensures consistency in the variations of the same word.
- **Comments parser**: Comments are read line by line and converted into structured columns.
- **Captions parser**: Small overlaps between consecutive lines are removed to avoid repetition that is found in my caption file. This is mostly the case for auto-generated captions.

The heads of cleaned captions and cleans comments are printed in the last creating dataframes.

Exporting Cleaned Data

Finally, both cleaned DataFrames were saved as **cleaned_comments.csv** and **cleaned_captions.csv** for reuse purposes.

Code Implementation

Parsing Comments

```
# Comment Parser
def structure_comments_from_txt(filepath='all_comments.txt'):
    with open(filepath, 'r', encoding='utf-8') as f:
        lines = f.readlines()

    comments_data = []

    # Each line looks like:
    # "1. @username [2025-10-19T22:03:01Z] 0 Likes → This was
good!"
    pattern = re.compile(
        r'^\s*\d+\.\s*@([\w\-\_]+)\s*\[([0-9T:\-
Z]+)\]\s*(\d+)\s*Likes?\s*→\s*(.*)$'
    )

    for line in lines:
        line = line.strip()
        match = pattern.match(line)
        if match:
            username = match.group(1)
            timestamp = match.group(2)
            likes = int(match.group(3))
            comment_text = match.group(4).strip()

            # Add length-based features
            char_length = len(comment_text)
            word_count = len(comment_text.split())
            avg_word_length = round(char_length / word_count,
2) if word_count else 0

            comments_data.append({
                'username': username,
                'timestamp_text': timestamp,
                'likes': likes,
                'comment_text': comment_text,
```

```
                'char_length': char_length,
                'word_count': word_count,
                'avg_word_length': avg_word_length
            })

    return pd.DataFrame(comments_data)

comments_df = structure_comments_from_txt()
comments_df.head()
```

Parsing Captions

```
# Comment Parser
def structure_comments_from_txt(filepath='all_comments.txt'):
    with open(filepath, 'r', encoding='utf-8') as f:
        lines = f.readlines()

    comments_data = []

    # Each line looks like:
    # "1. @username [2025-10-19T22:03:01Z] 0 Likes → This was
good!"
    pattern = re.compile(
        r'^\s*\d+\.\s*@([\w\-\_]+)\s*\[([0-9T:\-
Z]+)\]\s*(\d+)\s*Likes?\s*→\s*(.*)$'
    )

    for line in lines:
        line = line.strip()
        match = pattern.match(line)
        if match:
            username = match.group(1)
            timestamp = match.group(2)
            likes = int(match.group(3))
            comment_text = match.group(4).strip()

            # Add length-based features
            char_length = len(comment_text)
```

```
            word_count = len(comment_text.split())
            avg_word_length = round(char_length / word_count,
2) if word_count else 0

            comments_data.append({
                'username': username,
                'timestamp_text': timestamp,
                'likes': likes,
                'comment_text': comment_text,
                'char_length': char_length,
                'word_count': word_count,
                'avg_word_length': avg_word_length
            })

    return pd.DataFrame(comments_data)

comments_df = structure_comments_from_txt()
comments_df.head()
```

Cleaning Functions

```
# Cleaning pipeline
stop_words = set(stopwords.words('english'))
lemmatizer, stemmer = WordNetLemmatizer(), PorterStemmer()

def normalize_text(text: str) -> str:
    if text is None: return ""
    text = text.lower()
    text = re.sub(r'\[.*?\]', ' ', text)
    text = re.sub(r'[^a-z\s]', ' ', text)      # change to
[^a-z0-9\s] to keep digits
    text = re.sub(r'\s+', ' ', text).strip()
    return text

def tokenize_text(text: str):
    if not text: return []
    return word_tokenize(text)

def remove_stopwords(tokens):
```

```
    return [w for w in tokens if w not in stop_words and
len(w) > 2]

def lemmatize_tokens(tokens):
    return [lemmatizer.lemmatize(w) for w in tokens]

def stem_tokens(tokens):
    return [stemmer.stem(w) for w in tokens]

def clean_text_pipeline(text: str, use_lemmatization: bool =
True):
    x = normalize_text(text)
    toks = tokenize_text(x)
    toks = remove_stopwords(toks)
    return lemmatize_tokens(toks) if use_lemmatization else
stem_tokens(toks)

# Comments parser
def structure_comments_from_txt(filepath='all_comments.txt'):
    pat = re.compile(
        r'^\s*\d+\.\s*@([\w\-]+)\s*\[([0-9T:\-
Z]+)\]\s*([\d,]+)\s*Likes?\s*→\s*(.+)$'
    )
    rows = []
    with open(filepath, 'r', encoding='utf-8') as f:
        for raw in f:
            m = pat.match(raw.strip())
            if not m:
                continue
            username  = m.group(1)
            timestamp = m.group(2)
            likes = int(m.group(3).replace(',', ''))
            comment = m.group(4).strip()
            rows.append({'username': username,
                         'timestamp': timestamp,
                         'likes': likes,
                         'comment_text': comment})
    return pd.DataFrame(rows,
columns=['username','timestamp','likes','comment_text'])
```

```
# Captions parser
_SENT_SPLIT = re.compile(r'(?<=[.!?])\s+')

def _c(s: str) -> str:
    # minimal clean: strip, collapse spaces, drop stray quotes
    s = ' '.join(s.strip().split())
    return s.strip('"\'""“”‘’')

def _deoverlap(prev_line: str, curr_line: str, k: int = 6) ->
str:
    """
    Remove up to k-word overlap between the end of prev_line
and start of curr_line.
    This kills the rolling-window repetition common in auto-
captions.
    """
    a = prev_line.split()
    b = curr_line.split()
    for n in range(min(k, len(a), len(b)), 0, -1):
        if a[-n:] == b[:n]:
            return ' '.join(b[n:])
    return curr_line

def structure_captions_from_vtt(filepath='captions.en.vtt'):
    # read all cues (assumes webvtt works; no extra error
handling)
    cues = [_c(cue.text) for cue in WebVTT().read(filepath) if
cue.text.strip()]
    if not cues:
        return pd.DataFrame({'caption_sentence': []})

    # merge cues while removing small overlaps
    merged = [cues[0]]
    for t in cues[1:]:
        extra = _deoverlap(merged[-1], t, k=6)
        if extra:
            merged[-1] = merged[-1] + ' ' + extra
```

```
    # split into sentences
    big_text = ' '.join(merged)
    sentences = [s.strip() for s in
_SENT_SPLIT.split(big_text) if s.strip()]

    return pd.DataFrame({'caption_sentence': sentences})

# Preview
comments_df = structure_comments_from_txt('all_comments.txt')
comments_df['cleaned_tokens'] =
comments_df['comment_text'].apply(clean_text_pipeline)
print("Comments head:")
print(comments_df.head())

captions_df = structure_captions_from_vtt('captions.en.vtt')
captions_df['cleaned_tokens'] =
captions_df['caption_sentence'].apply(clean_text_pipeline)
print("\nCaptions head:")
print(captions_df.head())
```

Results and Discussion

In this I will analyze and discuss the outputs of the experiments I performed on my created YouTube dataset. It will also include the discussion about the challenges I faced during the whole process.

Dataset Overview

A total of **963 comments** and **62 captions** were selected for final observation. Each of these rows have gone through under rigorous cleaning and preprocessing process for a quality analysis.

Tokenization Examples

After removal of stopwords to get more impactful words-

- **Comments** were transformed into key tokens.

  Example: “this was good, but probably needs to be updated…”

→ ['good', 'probably', 'need', 'updated', 'example', 'used', 'spotify', 'amp', 'twilio', 'outdated']

- **Captions** were similarly created into shorter keyword lists.

  Example: “this course is an excellent introduction to apis…”

  → ['course', 'excellent', 'introduction', 'apis', 'beginner', 'updated', 'api']

Stopword Removal

A total of **69,519 stopwords** were removed, about **9,210** from comments and **60,309** from captions. Removing them helped clean up the text so that only the most meaningful words remained for analysis. This allowed the results to focus more on what people were talking about, rather than on everyday connecting words.

Frequent Tokens

The top 10 most common tokens in the **comments** were mainly web and tech-related terms such as **‘http’ (293)**, **‘href’ (289)**, **‘watch’ (289)**, **‘youtube’ (283)**, **‘com’ (275)**, **‘amp’ (274)**, **‘www’ (272)**, **‘wxsd’ (268)**, **‘zgxjrw’ (268)**, and **‘api’ (201)**.

On the other hand, **captions** most frequent words reflect a conversational tone. The top tokens included **‘going’ (1878)**, **right’ (701)**, **‘see’ (641)**, **‘let’ (620)**, **‘get’ (606)**, **‘message’ (567)**, **‘want’ (546)**, **‘api’ (521)**, **‘like’ (514)**, and **‘one’ (484)**. These words point to explanations, demonstrations, and learner-focused phrasing commonly found in teaching materials or course subtitles.

Data Quality Check

We can observe minor irregularities through the results. Firstly, no null text entries were found in both comments and captions files. However, **163 comments** contained non-ASCII characters and about **400** had multiple symbols. Although captions had **no non-ASCII** lines, **7 symbols** were even detected here.

Engagement Analysis

Correlations between comment features and the number of likes were weak but positive:

- Likes vs. word length → **0.029**

- Likes vs. token length → **0.013**
- Likes vs. character length → **0.015**

This suggests that longer comments did not necessarily receive more engagement.

Most-Liked Comments

The top three most-liked comments were by:

1. **SelahadinJemal** — 1976 likes
2. **sohailahmedalvi** — 507 likes
3. **magtegaaaaa** — 221 likes

These comments reflected positive learner sentiment and appreciation for accessible teaching.

Challenges Faced and Solutions

The lab was fairly straightforward and learner friendly. Although I did not encounter any major challenges, a small anomaly was noticed in the comment section. For example, the number of likes in a particular comment from **1965 to 1976** and another comment from **505 to 507**. The change was also noticed in the total number of comments fetched which rose from **980** to **983**. Although these variations did not cause any issues with the analysis, it served as a useful reminder that real-world data can change dynamically and may not always stay consistent between runs.

Conclusion

This lab gave me a good understanding of text cleaning works in real time data. Using asvace language processing tools like NLTK made the process a lot smoother and more efficient compared to the manual methods used in the previous lab.

I also watch how real-life data can change in a matter of time. Small changes like the number of likes or comments can happen anytime. Overall, the lab helped me see how important preprocessing is in text mining and how it sets the foundation for accurate and meaningful analysis for creating LLM models.