

whatsapp_chat_analyzer

1. Overview

The **WhatsApp Chat Analyzer** is a data analysis and visualization project that processes exported WhatsApp chats (group or personal) and generates insightful summaries about user behavior, activity patterns, and messaging trends.

Users can export any WhatsApp chat text file and upload it to the deployed web application. The application then performs both **overall group-level analysis** and **individual user-level analysis** and displays results in an interactive dashboard.

This project involves:

1. **Chat Analysis** – Extracting and processing text data from exported WhatsApp chats.
2. **Web Application** – Building a user interface to upload chats and display visual analytics.
3. **Deployment** – Hosting the full project on **Heroku** for public access.

Features Shown on the Website

1. Group-Level Analysis

A. Top Statistics

The dashboard displays the overall statistics of the chat:

- **Total Messages**
- **Total Words**
- **Number of Media Files Shared**
- **Number of Links Shared**

B. Monthly Timeline

A month-by-month graph showing:

- Messaging trends across months
 - Peaks and drops in activity
 - Seasonal or event-based patterns
-

C. Daily Timeline

A day-wise line graph showing:

- How message frequency varies day to day
 - Identification of active days and low-activity days
-

D. Activity Map

Visual insights into chat activity:

- **Most Busy Day**
 - **Most Busy Month**
 - Patterns in user engagement over time
-

E. Weekly Activity Heatmap

A heatmap representing:

- Hourly activity across days of the week
 - Helps identify peak chatting hours (e.g., late night, morning)
-

F. Most Busy Users (For Group Chats)

A ranking chart that shows:

- Users who send the most messages
 - Contribution percentage of each member
 - Useful for understanding dominant participants in groups
-

G. Word Cloud

A creative text visualization displaying:

- Most frequently used words in the chat
 - Larger words indicate higher frequency
-

H. Most Common Words

A detailed table or bar chart showing:

- Exact count of top frequently used words
 - Helps identify conversation topics and patterns
-

I. Emoji Analysis

Analysis of emojis used in the chat:

- Most frequently used emojis
 - Total count of emojis
 - Emoji distribution chart
-



2. Individual User-Level Analysis

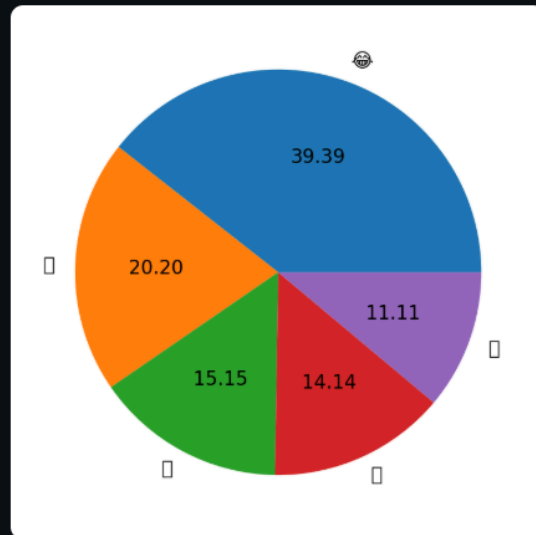
The user can select a specific person from the chat to see:

- Their total messages
- Their most used words
- Their emojis
- Their monthly and daily timelines
- Personalized activity map
- Their contribution to the group

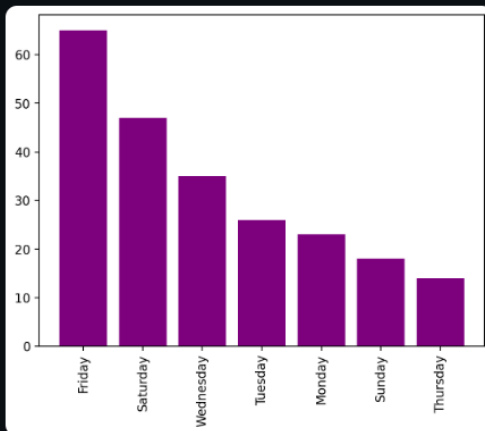
This allows users to understand individual behavior and patterns.

Emoji Analysis

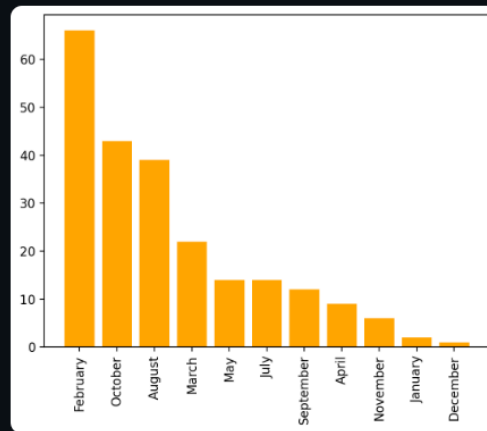
	0	1
0	👉	39
1	👉	20
2	👉	15
3	👉	14
4	👉	11
5	👉	10
6	👉	9
7	✓	9
8	👉	7
9	✓	7



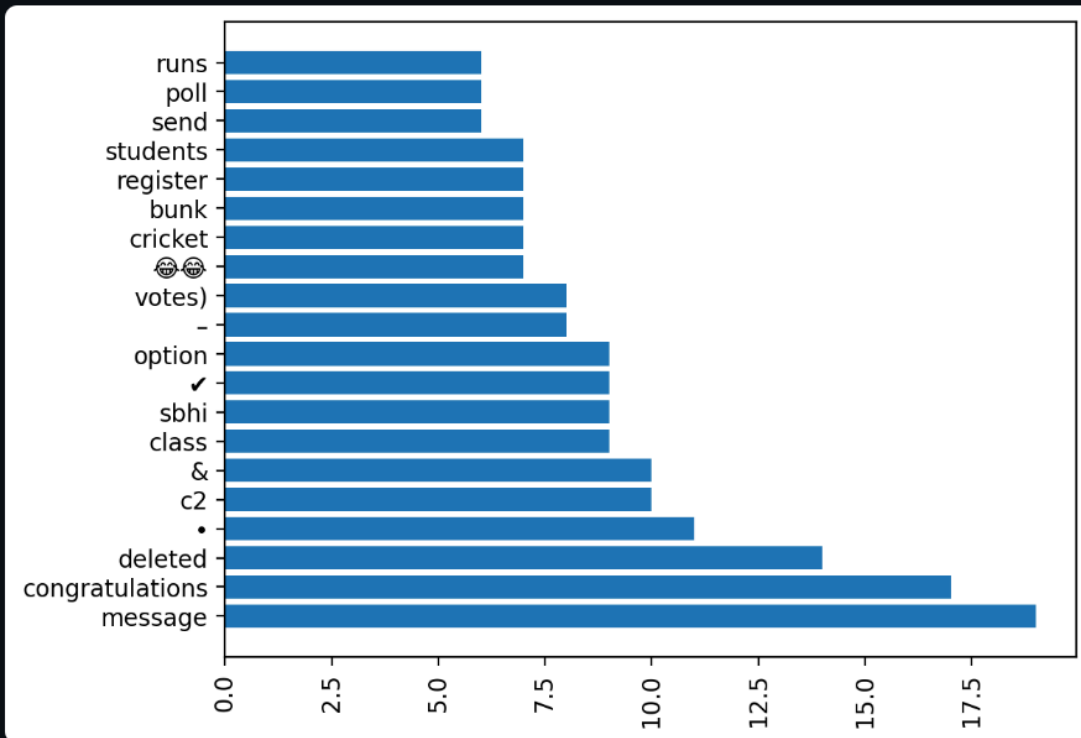
Most Busy Day



Most Busy Month



Most Common Words



2. PROJECT SETUP

Jupyter Notebook Explanation

1. Notebook Overview

This notebook is the core of the **WhatsApp Chat Analyzer** project. It includes:

1. Importing necessary libraries

For data handling, visualization, text processing, and date/time analysis.

2. Loading WhatsApp chat data

Reading the exported `.txt` chat file and converting it into a structured dataframe.

3. Preprocessing the chat data

- Extracting timestamps, users, and messages

- Handling media messages, links, and emojis
 - Cleaning text for analysis
 - 4. Generating basic statistics**
 - Total messages, total words, media shared, and links
 - 5. Timeline analysis**
 - Monthly timeline graph
 - Daily timeline graph
 - 6. Activity analysis**
 - Weekly activity heatmap
 - Most busy days and months
 - 7. User analysis (for groups)**
 - Most active users
 - Messages per user
 - Percentage contribution
 - 8. Text analysis**
 - Word cloud of frequently used words
 - Most common words table
 - 9. Emoji analysis**
 - Count and visualize emoji usage
 - 10. Individual user analysis**
 - Similar metrics as group analysis but filtered for a single person
 - 11. Visualization**
 - Graphs, charts, and plots for easy understanding
-

2. Code Sections Explained

A. Import Libraries

Typical libraries include:

- `pandas` → For data manipulation
- `numpy` → For numerical operations
- `matplotlib` / `seaborn` → For plotting graphs
- `re` → For regular expressions to parse text
- `emoji` → For emoji detection
- `wordcloud` → For generating word clouds
- `datetime` → For date-time processing

Purpose: Prepare the environment for data extraction, analysis, and visualization.

B. Load Chat Data

- Chat file is read as a text file.
 - Messages are split based on **timestamps** (usually WhatsApp format: `dd/mm/yyyy, hh:mm - user: message`).
 - A **DataFrame** is created with columns like:
 - `Date` / `Time`
 - `User`
 - `Message`
-

C. Preprocessing

Key steps:

1. Extract date, time, user, and message

Using regex to separate metadata from the actual message.

2. Handle media messages

Replace `<Media omitted>` with a placeholder.

3. Extract links

Identify URLs in messages for link count.

4. Clean messages

Remove unnecessary spaces, punctuation, or special characters if required.

D. Statistics

- **Total messages** → Number of rows in DataFrame
 - **Total words** → Sum of word counts per message
 - **Media shared** → Count of `<Media omitted>` entries
 - **Links shared** → Count of URLs in messages
-

E. Timeline Analysis

1. Monthly Timeline:

- Group messages by month and count frequency
- Plot line chart

2. Daily Timeline:

- Count messages per day
 - Identify active days
-

F. Activity Map

- **Most Busy Day & Month:** Using groupby on weekday/month
 - **Weekly Heatmap:** Messages per hour vs day of week to visualize peak activity
-

G. User Analysis

- For **group chats**, count:
 - Messages per user
 - Contribution percentage

- Identify most active users
-

H. Word Cloud and Common Words

- **Word Cloud:** Visual representation of most frequently used words
 - **Most Common Words:** Table with word counts
 - Stopwords (like "the", "is") are removed to focus on meaningful words
-

I. Emoji Analysis

- Detect all emojis in messages
 - Count frequency
 - Plot top emojis used
 - Helpful to understand emotional content of chats
-

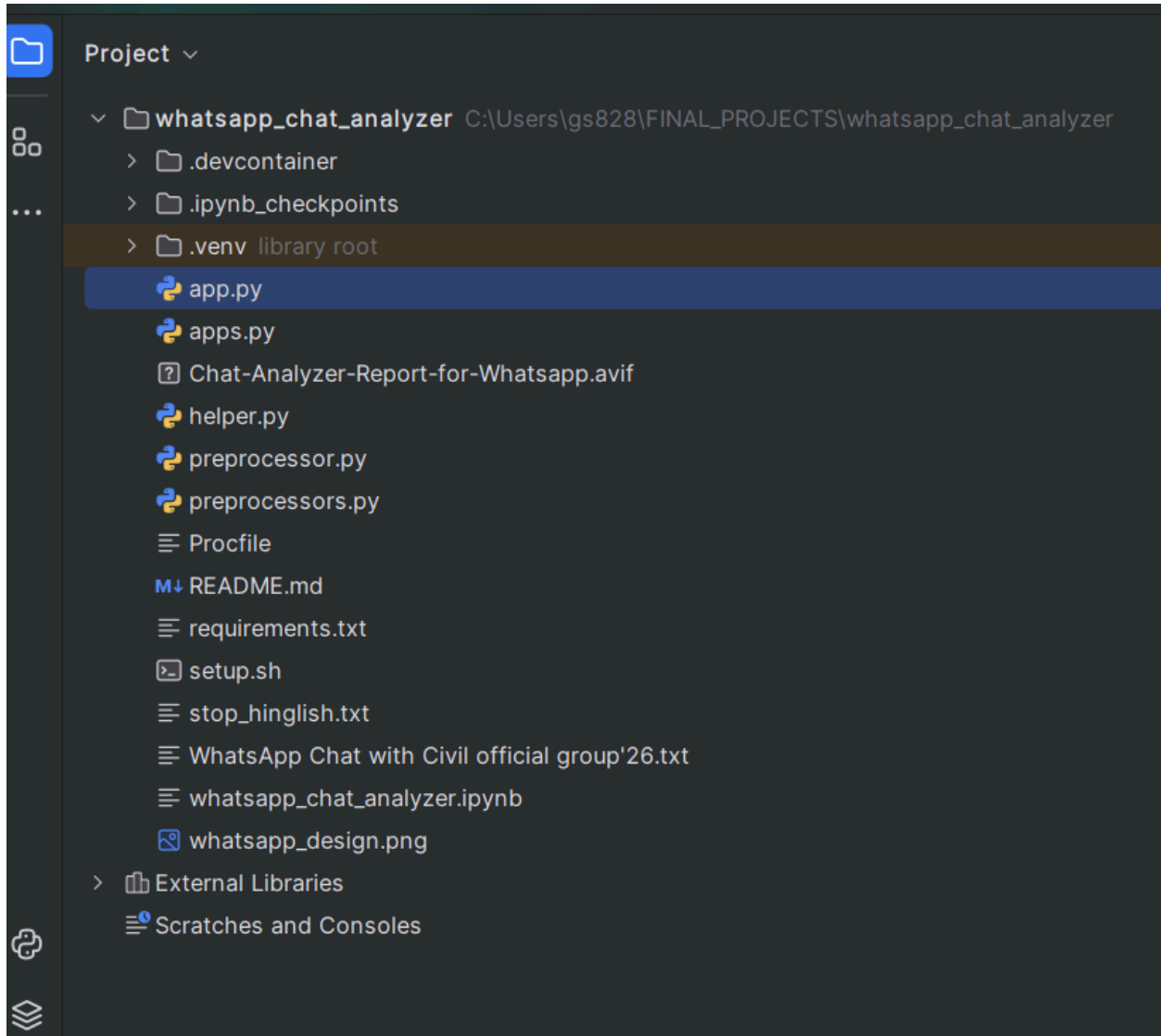
J. Individual User Analysis

- Filter DataFrame for a selected user
 - Compute:
 - Total messages
 - Words used
 - Emojis used
 - Timeline and activity patterns
-

K. Visualization

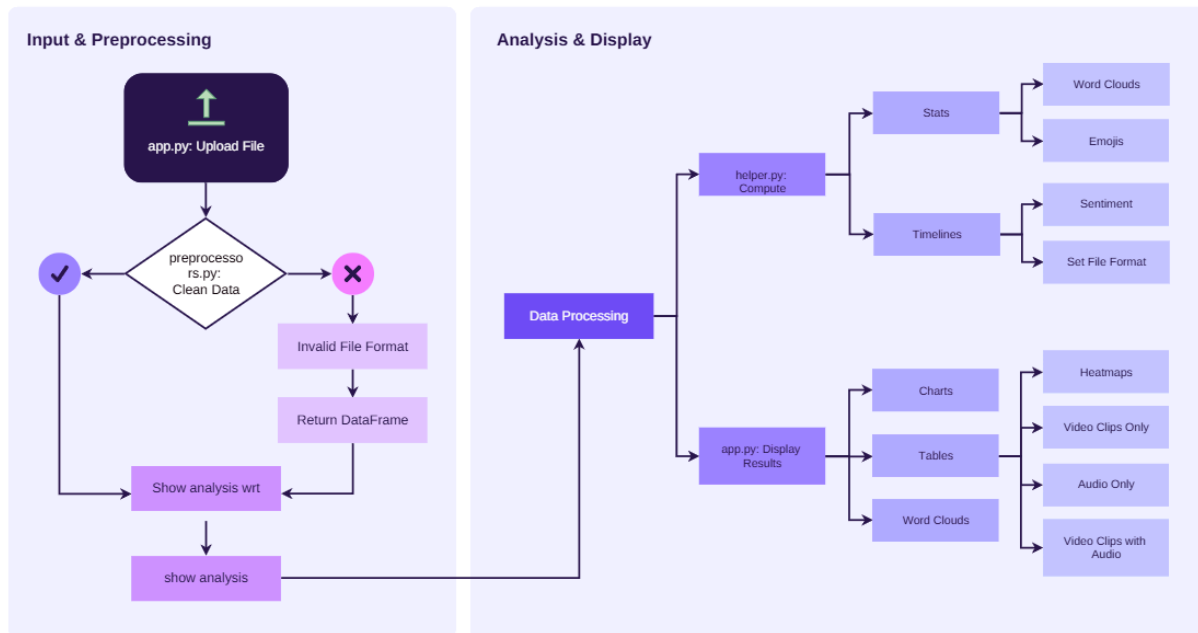
- Seaborn and Matplotlib used for:
 - Line plots (timelines)
 - Heatmaps (activity map)
 - Bar charts (most active users)
 - Word clouds (text analysis)

3. Initializing the project in Pycharm

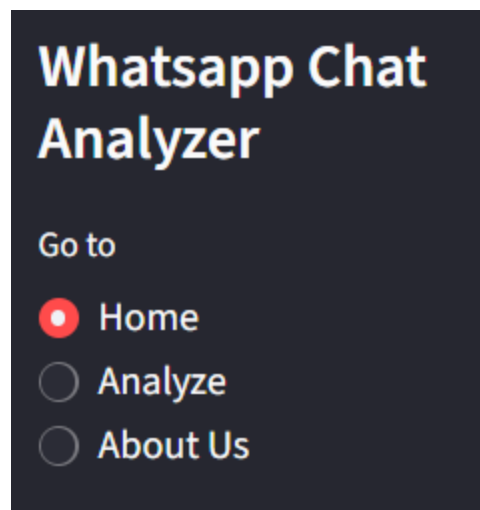


Python App Data Flow

Data Analyzer



Define Section Markdown in `app.py` streamlit



The **WhatsApp Chat Analyzer** is a powerful tool that helps you gain insights from your WhatsApp chat data. By simply uploading your chat file, you can explore various metrics and visualizations that reveal patterns and trends in your conversations.

Features

- **Total Messages:** Count the total number of messages exchanged in a chat.
 - **Word Analysis:** Analyze the total words used and get insights into your vocabulary.
 - **Media Analysis:** Count the number of media files shared (images, videos, audio, etc.).
 - **Link Analysis:** Identify how many links were shared in your chats.
 - **Sentiment Analysis:** Understand the overall sentiment (positive, negative, neutral) of the conversation.
 - **Emoji Analysis:** Track and visualize the emojis used in the chat.
 - **Monthly Timeline:** Visualize how many messages were sent each month.
 - **Daily Timeline:** See daily message trends and activity patterns.
 - **Activity Maps:** Discover the most active days of the week and months.
 - **Word Clouds:** Generate word clouds to see the most frequently used words.
 - **Common Words:** Identify the top words used in the chat.
 - **Busy Users:** Find out who is the most active user in a group chat.
-

How to Use It

1. **Upload File:** Choose your WhatsApp chat file (in `.txt` format) from the sidebar.
 2. **Select User:** Select the user for whom you want the analysis, or choose **Overall** for a group-wide analysis.
 3. **View Analysis:** Click **Show Analysis** to view detailed insights, visualizations, and statistics.
-

Restrictions

- The app currently supports only `.txt` files exported from WhatsApp.
 - Very large files may affect performance, so avoid uploading excessively large chats.
-

Memory and Performance

- The app can efficiently handle files up to **200MB**.
- Files larger than 200MB may take longer to process and could slow down the app.

Privacy and Security

- **Data Privacy:** Your uploaded chat data is processed locally and is **not stored on any server**.
- **Security:** The app does **not share your data** with any third parties. All processing is done **in-memory**, and data is discarded after your session ends.

A. **app.py** Explanation(12-Hour Version)

```
import streamlit as st
import preprocessor, helper
import matplotlib.pyplot as plt
import seaborn as sns
```

```
# Define sections
```

```
def home():
```

```
    st.title("Welcome to WhatsApp Chat Analyzer")
```

```
    st.markdown("""
```

```
        ## Product Description
```

The WhatsApp Chat Analyzer is a powerful tool designed to help you gain insights from your WhatsApp chat data. By simply uploading your chat file, you can explore various metrics and visualizations that reveal patterns and trends in your conversations.

```
        ## Features
```

- ****Total Messages****: Count the total number of messages exchanged.
- ****Word Analysis****: Analyze the total words used in the chat.

- **Media Analysis**: Count the number of media files shared.
- **Link Analysis**: Count the number of links shared.
- **Sentiment Analysis**: Understand the sentiment of the conversation.
- **Emoji Analysis**: Analyze the emojis used in the chat.
- **Monthly Timeline**: Visualize the number of messages sent each month.
- **Daily Timeline**: See daily message trends.
- **Activity Maps**: Discover the most active days and months.
- **Word Clouds**: Generate word clouds to see the most frequently used words.
- **Common Words**: Identify the most common words in the chat.
- **Busy Users**: Find out who is the most active user in the group.

How to Use It

1. **Upload File**: Choose your WhatsApp chat file (in `.txt` format) from the sidebar.
2. **Select User**: Select the user for whom you want to see the analysis or choose 'Overall' for group analysis.
3. **View Analysis**: Click on 'Show Analysis' to view detailed insights and visualizations.

Restrictions

- The app currently supports only `.txt` files exported from WhatsApp.
- Ensure the chat file is not too large to avoid performance issues.

Memory Size

- The app can handle files up to 200MB efficiently.
- Larger files may take more time to process and could impact performance.

Privacy and Security

- **Data Privacy**: Your uploaded chat data is processed locally and not stored on any server.
- **Security**: The app does not share your data with third parties. All processing is done in-memory and the data is discarded after the session ends.

```

    Enjoy analyzing your WhatsApp conversations with our interactive tools!
    """)

# Add images
image_path1 = r"C:\Users\gs828\FINAL_PROJECTS\whatsapp_chat_analyze
r\Chat-Analyzer-Report-for-Whatsapp.avif"
image_path = r"C:\Users\gs828\FINAL_PROJECTS\whatsapp_chat_analyzer
\whatsapp_design.png" # Update with the correct path to your image
st.image(image_path, caption="WhatsApp Chat Analysis", width=400)
st.image(image_path1, caption="WhatsApp Chat Analysis", width=400)

def analyze():
    st.title("WhatsApp Chat Analysis")
    st.sidebar.title("WhatsApp Chat Analyzer")
    uploaded_file = st.sidebar.file_uploader("Choose a file")

    if uploaded_file is not None:
        bytes_data = uploaded_file.getvalue()
        data = bytes_data.decode("utf-8")
        df = preprocessor.preprocess(data)

        # fetch unique users
        user_list = df['user'].unique().tolist()
        user_list.remove('group_notification')
        user_list.sort()
        user_list.insert(0, "Overall")

        selected_user = st.sidebar.selectbox("Show analysis wrt", user_list)

        if st.sidebar.button("Show Analysis"):
            # Stats Area
            num_messages, words, num_media_messages, num_links = helper.fetc
h_stats(selected_user, df)
            st.title("Top Statistics")
            col1, col2, col3, col4 = st.columns(4)

```

```

with col1:
    st.header("Total Messages")
    st.title(num_messages)
with col2:
    st.header("Total Words")
    st.title(words)
with col3:
    st.header("Media Shared")
    st.title(num_media_messages)
with col4:
    st.header("Links Shared")
    st.title(num_links)

st.title("Sentiment Analysis")
sentiment_df = helper.sentiment_analysis(selected_user, df)
fig, ax = plt.subplots()
ax.bar(sentiment_df['sentiment_label'], sentiment_df['message'], color
= ['red', 'blue', 'green'])
st.pyplot(fig)

# Emoji analysis
st.title("Emoji Analysis")
emoji_df = helper.emoji_helper(selected_user, df)
col1, col2 = st.columns(2)

with col1:
    st.dataframe(emoji_df)
with col2:
    fig, ax = plt.subplots()
    ax.pie(emoji_df[1].head(), labels=emoji_df[0].head(), autopct="%0.2
f")

    st.pyplot(fig)

# Monthly timeline
st.title("Monthly Timeline")
timeline = helper.monthly_timeline(selected_user, df)

```

```

fig, ax = plt.subplots()
ax.plot(timeline['time'], timeline['message'], color='green')
plt.xticks(rotation='vertical')
st.pyplot(fig)

# Daily timeline
st.title("Daily Timeline")
daily_timeline = helper.daily_timeline(selected_user, df)
fig, ax = plt.subplots()
ax.plot(daily_timeline['only_date'], daily_timeline['message'], color='black')

plt.xticks(rotation='vertical')
st.pyplot(fig)

# Activity map
st.title('Activity Map')
col1, col2 = st.columns(2)

with col1:
    st.header("Most Busy Day")
    busy_day = helper.week_activity_map(selected_user, df)
    fig, ax = plt.subplots()
    ax.bar(busy_day.index, busy_day.values, color='purple')
    plt.xticks(rotation='vertical')
    st.pyplot(fig)

with col2:
    st.header("Most Busy Month")
    busy_month = helper.month_activity_map(selected_user, df)
    fig, ax = plt.subplots()
    ax.bar(busy_month.index, busy_month.values, color='orange')
    plt.xticks(rotation='vertical')
    st.pyplot(fig)

st.title("Weekly Activity Map")
user_heatmap = helper.activity_heatmap(selected_user, df)

```

```

fig, ax = plt.subplots()
ax = sns.heatmap(user_heatmap)
st.pyplot(fig)

# Finding the busiest users in the group (Group level)
if selected_user == 'Overall':
    st.title('Most Busy Users')
    x, new_df = helper.most_busy_users(df)
    fig, ax = plt.subplots()

    col1, col2 = st.columns(2)

    with col1:
        ax.bar(x.index, x.values, color='red')
        plt.xticks(rotation='vertical')
        st.pyplot(fig)
    with col2:
        st.dataframe(new_df)

# WordCloud
st.title("Wordcloud")
df_wc = helper.create_wordcloud(selected_user, df)
fig, ax = plt.subplots()
ax.imshow(df_wc)
st.pyplot(fig)

# Most common words
most_common_df = helper.most_common_words(selected_user, df)
fig, ax = plt.subplots()
ax.barh(most_common_df[0], most_common_df[1])
plt.xticks(rotation='vertical')

st.title('Most Common Words')
st.pyplot(fig)

# Emoji analysis (repeated)

```

```

st.title("Emoji Analysis")
emoji_df = helper.emoji_helper(selected_user, df)
col1, col2 = st.columns(2)

with col1:
    st.dataframe(emoji_df)
with col2:
    fig, ax = plt.subplots()
    ax.pie(emoji_df[1].head(), labels=emoji_df[0].head(), autopct="%0.2
f")

    st.pyplot(fig)

def about_us():
    st.title("About Us")
    st.markdown("""
    ## About Our Team

    We are a group of passionate data scientists and software engineers dedi
    cated to making data analysis accessible and useful for everyone. Our Whats
    App Chat Analyzer project aims to provide users with insightful analytics from
    their chat data in an easy-to-use interface.

    ## Our Mission

    Our mission is to empower users to gain meaningful insights from their c
    onversations and help them understand their communication patterns better.

    ## Contact Us

    If you have any questions or feedback, feel free to reach out to us at:
    - Email: gauravsingh12430@gmail.com
    - Phone: 7818921547
    - Address: Jawahar Lal Nehru Marg, Jhalana Gram, Malviya Nagar, Jaipu
    r, Rajasthan 302017

    Follow us on social media:
    - Twitter: [@whatsapp_analyzer](https://twitter.com/whatsapp_analyzer)
    - LinkedIn: [WhatsApp Chat Analyzer](https://www.linkedin.com/compan
    y/whatsapp-chat-analyzer)

```

```

        Thank you for using our WhatsApp Chat Analyzer tool!
    """

# Sidebar navigation
st.sidebar.title("Whatsapp Chat Analyzer")
page = st.sidebar.radio("Go to", ["Home", "Analyze", "About Us"])

# Display the selected section
if page == "Home":
    home()
elif page == "Analyze":
    analyze()
elif page == "About Us":
    about_us()

```

Your Streamlit-based WhatsApp Chat Analyzer works by combining:

- **UI layer** → Streamlit
- **Data preprocessing** → `preprocessor.py`
- **Analysis functions** → `helper.py`
- **Visualizations** → Matplotlib + Seaborn

Below is a clear, section-by-section breakdown.

✓ 1. Importing Dependencies

```

import streamlit as st
import preprocessor, helper
import matplotlib.pyplot as plt
import seaborn as sns

```

Purpose of Each Import

Import	Purpose
<code>streamlit</code>	Creates the complete web application UI (buttons, sidebar, charts).
<code>preprocessor</code>	Custom module that converts raw WhatsApp chat data into a clean DataFrame (parsing date, time, users, messages).
<code>helper</code>	Performs analytics like stats counting, timelines, sentiment analysis, emoji analysis, etc.
<code>matplotlib.pyplot</code>	Used for line plots, bar charts, and pie charts in the analysis.
<code>seaborn</code>	Enhances plots (mostly heatmaps) for better visuals.

✓ 2. Home Page (`home()` Function)

This is the landing screen for users.

```
def home():
    st.title("Welcome to WhatsApp Chat Analyzer")
    st.markdown("""" ... """)
    st.image(image_path, width=400)
    st.image(image_path1, width=400)
```

✓ What the Home Page Does

1. Displays **welcome title**.
2. Shows **product description**.
3. Explains **all features** of the tool:
 - Total messages
 - Word count
 - Media count
 - Emoji count
 - Sentiment
 - Timelines

- Activity heatmaps
- Word cloud

4. Describes:

- **How to upload** WhatsApp chat files
- **Memory limit**
- **Privacy protection**

5. Shows **two images** for a professional UI.

The `home()` function is entirely UI-driven with no backend computations.

✓ 3. Analysis Page (`analyze()` Function)

This is the **core functionality** of your project.

```
def analyze():
    st.title("WhatsApp Chat Analysis")
```

✓ Sidebar Setup

```
st.sidebar.title("WhatsApp Chat Analyzer")
uploaded_file = st.sidebar.file_uploader("Choose a file")
```

- Sidebar presents the file uploader.
- Accepts `.txt` WhatsApp exports.

★ 3.1 Loading & Preprocessing the Chat File

```
bytes_data = uploaded_file.getvalue()
data = bytes_data.decode("utf-8")
```

```
df = preprocessor.preprocess(data)
```

✓ What happens here?

1. User uploads the `.txt` chat file
2. File is read as bytes
3. Decoded into a text string
4. Passed to `preprocessor.preprocess()`
5. Preprocessor extracts:
 - Date (with AM/PM)
 - Time
 - Username
 - Message text
6. Returns a clean **Pandas DataFrame** for analysis.

★ 3.2 User Selection Dropdown

```
user_list = df['user'].unique().tolist()
user_list.remove('group_notification')
user_list.sort()
user_list.insert(0, "Overall")
selected_user = st.sidebar.selectbox("Show analysis wrt", user_list)
```

✓ Purpose:

- Shows list of participants from the chat.
- Removes WhatsApp system messages (`group_notification`).
- Adds **Overall** option for group-level analysis.

- Allows user to analyze:
 - Entire group
 - A specific person

This drives all downstream analytics.

★ 4. Analysis Execution (Triggered by Button)

```
if st.sidebar.button("Show Analysis"):
```

When clicked, the following sections are displayed:

✓ 4.1 Statistics Overview

```
num_messages, words, num_media_messages, num_links = helper.fetch_stats  
(selected_user, df)
```

Displays 4 KPIs in columns:

Column	Shows
Total Messages	Number of messages sent
Total Words	Total words typed
Media Shared	Number of media files ("Media omitted")
Links Shared	Count of hyperlinks

These stats give a **quick snapshot of activity**.

✓ 4.2 Sentiment Analysis

```
sentiment_df = helper.sentiment_analysis(selected_user, df)
```

Shows a bar chart of:

- Positive messages
- Negative messages
- Neutral messages

This helps understand the **tone** of the conversation.

✓ 4.3 Emoji Analysis

```
emoji_df = helper.emoji_helper(selected_user, df)
```

Provides:

- Dataframe of emoji counts
- Pie chart of top emojis

Used to understand **engagement, emotions, and personality**.

✓ 4.4 Monthly Timeline

```
timeline = helper.monthly_timeline(selected_user, df)  
ax.plot(timeline['time'], timeline['message'])
```

Shows:

- Month-wise messaging trend
 - Helps visualize **consistency and spikes** in chat activity
-

✓ 4.5 Daily Timeline

```
daily_timeline = helper.daily_timeline(selected_user, df)
ax.plot(daily_timeline['only_date'], daily_timeline['message'])
```

Shows:

- Day-level activity
- Useful for finding specific **high-activity days**.

✓ 4.6 Activity Maps

Busy Day

```
busy_day = helper.week_activity_map(selected_user, df)
```

Shows which **day of the week** the user is most active.

Busy Month

```
busy_month = helper.month_activity_map(selected_user, df)
```

Shows which **month** has the most activity.

Weekly Heatmap

```
sns.heatmap(user_heatmap)
```

Shows:

- Hourly × weekday activity

- Highlights **peak usage times** visually.
-

✓ 4.7 Most Busy Users (Group Only)

```
if selected_user == 'Overall':  
    x, new_df = helper.most_busy_users(df)
```

Shows:

- Bar chart of most active users
- Table of user-level message counts

Useful for **group insights**.

✓ 4.8 WordCloud

```
df_wc = helper.create_wordcloud(selected_user, df)
```

Shows:

- Word cloud image
 - Highlights the **most frequently used words** visually.
-

✓ 4.9 Most Common Words

```
most_common_df = helper.most_common_words(selected_user, df)
```

Displays:

- Horizontal bar chart of top words

- Useful for understanding **conversation themes**
-

✓ 4.10 Final Emoji Analysis (Repeated)

Re-shown for end-of-report completeness.

✓ 5. About Us Page

Simple section explaining:

- Team
- Mission
- Contact info
- Social links

Gives the project a polished look.

★ 6. Sidebar Navigation

```
page = st.sidebar.radio("Go to", ["Home", "Analyze", "About Us"])
```

Controls which section is shown:

Option	Function
Home	Calls <code>home()</code>
Analyze	Calls <code>analyze()</code>
About Us	Calls <code>about_us()</code>

This is the **main navigation system** of the app.

🎯 7. Overall Workflow Summary

Here's how the user interacts:

1. Open app → Land on Home page
2. Upload WhatsApp chat `.txt`
3. Choose a user or group
4. Click "Show Analysis"
5. The app generates:
 - Stats
 - Sentiment
 - Emoji charts
 - Timelines
 - Activity patterns
 - Word cloud
 - Most common words
 - Most active users

The app is clean, modular, interactive, and designed for easy understanding.

B. `preprocessor.py` Explanation(12-Hour Version)

```
import re
import pandas as pd
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize

nltk.download('stopwords')
nltk.download('punkt')

def preprocess(data):
    pattern = r'\d{1,2}/\d{1,2}/\d{2,4},\s\d{1,2}:\d{2}\s[APMapm]{2}\s-\s'
```

```

messages = re.split(pattern, data)[1:]
dates = re.findall(pattern, data)

df = pd.DataFrame({'user_message': messages, 'message_date': dates})
# convert message_date type
df['message_date'] = pd.to_datetime(df['message_date'], format='%m/%
d/%y, %I:%M %p - ')

df.rename(columns={'message_date': 'date'}, inplace=True)

users = []
messages = []
for message in df['user_message']:
    entry = re.split('([\w\W]+?):\s', message)
    if entry[1]: # user name
        users.append(entry[1])
        messages.append(" ".join(entry[2:]))
    else:
        users.append('group_notification')
        messages.append(entry[0])

df['user'] = users
df['message'] = messages
df.drop(columns=['user_message'], inplace=True)

df['only_date'] = df['date'].dt.date
df['year'] = df['date'].dt.year
df['month_num'] = df['date'].dt.month
df['month'] = df['date'].dt.month_name()
df['day'] = df['date'].dt.day
df['day_name'] = df['date'].dt.day_name()
df['hour'] = df['date'].dt.hour
df['minute'] = df['date'].dt.minute

period = []
for hour in df[['day_name', 'hour']]['hour']:

```

```

if hour == 23:
    period.append(str(hour) + "-" + str('00'))
elif hour == 0:
    period.append(str('00') + "-" + str(hour + 1))
else:
    period.append(str(hour) + "-" + str(hour + 1))

df['period'] = period

return df

```

`preprocessor.py` is the **data-cleaning engine** of your WhatsApp Chat Analyzer.

Its job is to:

- Parse raw `.txt` WhatsApp chats
- Extract timestamps, usernames, messages
- Convert everything into a structured **DataFrame**
- Add useful time-based columns (month, hour, weekday...)

This prepares the data for advanced analysis in `helper.py` and visualizations in `app.py`.

✓ 1. Importing Necessary Libraries

```

import re
import pandas as pd
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize

nltk.download('stopwords')
nltk.download('punkt')

```

Why these libraries?

Library	Purpose
<code>re</code>	Regex used to detect timestamps and split messages.
<code>pandas</code>	Creates DataFrames for structured analysis.
<code>nltk</code>	Text processing library (stopwords, tokenization).
<code>stopwords</code>	Removes common meaningless words (future use).
<code>punkt</code>	Allows splitting sentences into tokens.

Note: Although NLTK is imported, the stopwords/tokenizer are used mainly inside helper.py for word-level analysis.

✓ 2. Main Function: `preprocess(data)`

```
def preprocess(data):
```

This function takes **raw chat text** and outputs a clean DataFrame ready for analysis.

★ Step-by-Step Breakdown of Preprocessing

3. Extracting Dates & Messages

```
pattern = r'\d{1,2}/\d{1,2}/\d{2,4},\s\d{1,2}:\d{2}\s[APMapm]{2}\s-\s'  
messages = re.split(pattern, data)[1:]  
dates = re.findall(pattern, data)
```

✓ What this regex matches

Example WhatsApp format (12-hour):

12/25/23, 8:15 PM -
01/10/2024, 11:02 AM -

✓ Explanation

- `re.split(pattern, data)` → splits the chat into message blocks
- `re.findall(pattern, data)` → extracts all timestamps
- `[1:]` removes first empty item because split begins before first match

This gives **two aligned lists**:

Index	Timestamp	Message
0	date1	message1
1	date2	message2
...

These are ready to combine into a DataFrame.

4. Creating the Initial DataFrame

```
df = pd.DataFrame({'user_message': messages, 'message_date': dates})
df['message_date'] = pd.to_datetime(df['message_date'],
                                   format='%m/%d/%y, %l:%M %p - ')
df.rename(columns={'message_date': 'date'}, inplace=True)
```

✓ What happens?

1. Create a DataFrame with two columns:

- `user_message` → raw message text
- `message_date` → raw date string

2. Convert timestamp from string → Python datetime
3. Rename column to `date` for easier handling

Example conversion:

"12/25/23, 8:15 PM -" → datetime object

This enables powerful time-based operations.

★ 5. Splitting Usernames From Messages

```
users = []
messages = []
for message in df['user_message']:
    entry = re.split('([\w\W]+?):\s', message)
    if entry[1:]: # user name
        users.append(entry[1])
        messages.append(" ".join(entry[2:]))
    else:
        users.append('group_notification')
        messages.append(entry[0])
```

✓ WhatsApp message structure:

User Name: message text

✓ Why this is needed?

Some lines are:

- User messages

- System messages (no username), for example:
 - "Messages to this group are now secured with end-to-end encryption."
 - "Gaurav joined using invite link"

✓ Logic:

- `entry[1]` = username
- `entry[2:]` = actual message
- If no username → label as `'group_notification'`

This cleans the data and separates **who said what**.

★ 6. Adding Extracted Columns Into DataFrame

```
df['user'] = users
df['message'] = messages
df.drop(columns=['user_message'], inplace=True)
```

✓ Resulting DataFrame now has:

date	user	message
------	------	---------

Clean and ready for analysis.

★ 7. Creating Powerful Time-Based Features

```
df['only_date'] = df['date'].dt.date
df['year'] = df['date'].dt.year
df['month_num'] = df['date'].dt.month
```

```
df['month'] = df['date'].dt.month_name()
df['day'] = df['date'].dt.day
df['day_name'] = df['date'].dt.day_name()
df['hour'] = df['date'].dt.hour
df['minute'] = df['date'].dt.minute
```

✓ Why create these columns?

These enable:

Column	Used For
year	yearly trends
month / month_name	monthly timeline
day_name	weekly activity charts
hour	hourly heatmap
only_date	daily timeline
minute	seldom used, kept for completeness

These features power the visualizations in [helper.py](#) and [app.py](#).

★ 8. Creating the Hourly “Period” Column

```
period = []
for hour in df[['day_name', 'hour']]['hour']:
    if hour == 23:
        period.append(str(hour) + "-" + str('00'))
    elif hour == 0:
        period.append(str('00') + "-" + str(hour + 1))
    else:
        period.append(str(hour) + "-" + str(hour + 1))
```

```
df['period'] = period
```

✓ Example

```
8 → "8-9"  
14 → "14-15"  
23 → "23-00"
```

✓ Purpose:

Used for:

- **Hourly activity heatmaps**
- **Time distribution plots**

Creates consistent interval representation.



9. Returning the Cleaned DataFrame

```
return df
```

The final DataFrame contains all cleaned and engineered features:

Column	Meaning
date	full datetime
user	sender
message	message text
only_date	date only
year, month_num, month	time categorization
day, day_name	daily activity
hour, minute	hour-level activity

Column	Meaning
period	time interval block

This DataFrame powers every visualization in your Streamlit web app.

Summary of What `preprocessor.py` Achieves

- Converts raw chat into clean, structured data
- Extracts usernames, messages, and timestamps
- Handles system messages
- Adds deep feature engineering (year, month, weekday, hour, period)
- Returns a ready-to-analyze DataFrame

OR

A. `apps.py` Explanation(24-Hour Version)

```
import streamlit as st
import preprocessors, helper
import matplotlib.pyplot as plt
import seaborn as sns

# =====
# HOME PAGE
# =====

def home():
    st.title("Welcome to WhatsApp Chat Analyzer (24-Hour)")
    st.markdown("""
        This version is configured for **24-hour timestamp WhatsApp chats**.
        Upload your .txt file and start analyzing messages, media, links,
        sentiment, emojis, weekly/monthly activity, and more.
```

```

"""
# =====
# ANALYSIS PAGE
# =====
def analyze():
    st.title("WhatsApp Chat Analysis (24-Hour Format)")
    st.sidebar.title("WhatsApp Chat Analyzer")

    uploaded_file = st.sidebar.file_uploader("Choose a WhatsApp Chat File (.txt)")

    if uploaded_file is not None:
        data = uploaded_file.getvalue().decode("utf-8")

        # use 24-hour preprocessors file
        df = preprocessors.preprocess(data)

        # fetch users
        user_list = df['user'].unique().tolist()
        if 'group_notification' in user_list:
            user_list.remove('group_notification')
        user_list.sort()
        user_list.insert(0, "Overall")

        selected_user = st.sidebar.selectbox("Show analysis wrt", user_list)

        if st.sidebar.button("Show Analysis"):

            # =====
            # Top Statistics
            # =====
            num_messages, words, num_media, num_links = helper.fetch_stats(selected_user, df)

            st.title("Top Statistics")

```

```

col1, col2, col3, col4 = st.columns(4)

with col1:
    st.header("Total Messages")
    st.title(num_messages)
with col2:
    st.header("Total Words")
    st.title(words)
with col3:
    st.header("Media Shared")
    st.title(num_media)
with col4:
    st.header("Links Shared")
    st.title(num_links)

# =====
# Sentiment Analysis
# =====
st.title("Sentiment Analysis")
sentiment_df = helper.sentiment_analysis(selected_user, df)
fig, ax = plt.subplots()
ax.bar(sentiment_df['sentiment_label'], sentiment_df['message'])
st.pyplot(fig)

# =====
# Emoji Analysis
# =====
st.title("Emoji Analysis")
emoji_df = helper.emoji_helper(selected_user, df)
col1, col2 = st.columns(2)

with col1:
    st.dataframe(emoji_df)
with col2:
    fig, ax = plt.subplots()
    ax.pie(emoji_df[1].head(), labels=emoji_df[0].head(), autopct="%0.2

```

```

f")

    st.pyplot(fig)

    # =====
    # Monthly Timeline
    # =====
    st.title("Monthly Timeline")
    timeline = helper.monthly_timeline(selected_user, df)
    fig, ax = plt.subplots()
    ax.plot(timeline['time'], timeline['message'])
    plt.xticks(rotation='vertical')
    st.pyplot(fig)

    # =====
    # Daily Timeline
    # =====
    st.title("Daily Timeline")
    daily = helper.daily_timeline(selected_user, df)
    fig, ax = plt.subplots()
    ax.plot(daily['only_date'], daily['message'])
    plt.xticks(rotation='vertical')
    st.pyplot(fig)

    # =====
    # Activity Maps
    # =====
    st.title('Activity Map')
    c1, c2 = st.columns(2)

    with c1:
        st.header("Most Busy Day")
        busy_day = helper.week_activity_map(selected_user, df)
        fig, ax = plt.subplots()
        ax.bar(busy_day.index, busy_day.values)
        plt.xticks(rotation='vertical')
        st.pyplot(fig)

```

```

with c2:
    st.header("Most Busy Month")
    busy_month = helper.month_activity_map(selected_user, df)
    fig, ax = plt.subplots()
    ax.bar(busy_month.index, busy_month.values)
    plt.xticks(rotation='vertical')
    st.pyplot(fig)

# Heatmap
st.title("Weekly Activity Heatmap")
heatmap = helper.activity_heatmap(selected_user, df)
fig, ax = plt.subplots()
sns.heatmap(heatmap, ax=ax)
st.pyplot(fig)

# =====
# Most Busy Users
# =====
if selected_user == "Overall":
    st.title("Most Busy Users")
    x, new_df = helper.most_busy_users(df)

    c1, c2 = st.columns(2)
    with c1:
        fig, ax = plt.subplots()
        ax.bar(x.index, x.values)
        plt.xticks(rotation='vertical')
        st.pyplot(fig)
    with c2:
        st.dataframe(new_df)

# =====
# Wordcloud
# =====
st.title("Wordcloud")

```

```

wc = helper.create_wordcloud(selected_user, df)
fig, ax = plt.subplots()
ax.imshow(wc)
st.pyplot(fig)

# =====
# Most Common Words
# =====
st.title("Most Common Words")
mc = helper.most_common_words(selected_user, df)
fig, ax = plt.subplots()
ax.barh(mc[0], mc[1])
plt.xticks(rotation='vertical')
st.pyplot(fig)

# =====
# ABOUT
# =====
def about_us():
    st.title("About Us - 24 Hour Version")
    st.write("Built by Gaurav Singh. Enhanced to support both YY and YYYY for mats.")

# =====
# SIDEBAR NAVIGATION
# =====
st.sidebar.title("WhatsApp Chat Analyzer")
page = st.sidebar.radio("Go to", ["Home", "Analyze", "About Us"])

if page == "Home":
    home()
elif page == "Analyze":
    analyze()

```

```
else:  
    about_us()
```

The `apps.py` file is the **main Streamlit application** for analyzing WhatsApp chats using **24-hour timestamps**. It controls the UI, interacts with preprocessing and analysis modules, and displays all analytical charts.

This version of the app uses:

- `preprocessors.py` → to parse 24-hour WhatsApp chat format
- `helper.py` → to generate all statistics, timelines, heatmaps, sentiment, and word analysis

✓ 1. Imports

```
import streamlit as st  
import preprocessors, helper  
import matplotlib.pyplot as plt  
import seaborn as sns
```

✓ Purpose of each library:

Module	Reason Used
<code>streamlit</code>	Creates UI components, layouts, buttons, sidebar, and displays charts.
<code>preprocessors</code>	Cleans raw WhatsApp text exported in 24-hour format .
<code>helper</code>	Contains all analysis functions like stats, timelines, heatmaps, wordcloud, sentiment, etc.
<code>matplotlib.pyplot</code>	Plots bar graphs, line charts, and pie charts.
<code>seaborn</code>	Used for visually appealing heatmaps (weekly activity).

✓ 2. Home Page — `home()`

```
def home():  
    st.title("Welcome to WhatsApp Chat Analyzer (24-Hour)")
```

✓ Purpose:

- Shows the landing page when the user opens the app.
- Provides a brief description of the tool.
- Highlights that this version is configured for **24-hour chat data**.

This page is simple, clean, and informative.

★ 3. Analysis Page — `analyze()`

This is the **heart** of the WhatsApp Chat Analyzer.

```
def analyze():  
    st.title("WhatsApp Chat Analysis (24-Hour Format)")
```

◆ 3.1 Sidebar Setup

```
st.sidebar.title("WhatsApp Chat Analyzer")  
uploaded_file = st.sidebar.file_uploader("Choose a WhatsApp Chat File (.txt)")
```

What this does:

- Displays a sidebar title.
- Adds a file uploader to allow users to upload `.txt` WhatsApp chats.

◆ 3.2 Reading & Preprocessing Chat Data

```
data = uploaded_file.getvalue().decode("utf-8")
df = preprocessors.preprocess(data)
```

- Extracts **raw text** from uploaded file.
- Passes it to `preprocessors.preprocess()` which:
 - Parses timestamps (24-hour)
 - Extracts user and messages
 - Adds engineered features (month, weekday, hour, period...)

The result is a **clean DataFrame**, ready for analysis.

◆ 3.3 User Selection Dropdown

```
user_list = df['user'].unique().tolist()
if 'group_notification' in user_list:
    user_list.remove('group_notification')
user_list.sort()
user_list.insert(0, "Overall")
selected_user = st.sidebar.selectbox("Show analysis wrt", user_list)
```

✓ What this does:

- Extracts all unique users from the chat.
- Removes WhatsApp automated system messages.
- Adds **Overall** option → for group-level analysis.
- Sidebar dropdown lets the user choose which user to analyze.

★ 4. Start Analysis (When Button Is Clicked)

```
if st.sidebar.button("Show Analysis"):
```

– When the user clicks this button → all charts and insights are generated.

★ 5. Section-by-Section Analysis

🔥 5.1 Top Statistics

```
num_messages, words, num_media, num_links = helper.fetch_stats(selected_user, df)
```

Displays 4 important metrics:

1. **Total Messages**
2. **Total Words**
3. **Media Shared**
4. **Links Shared**

They are displayed using 4 Streamlit columns:

```
col1, col2, col3, col4 = st.columns(4)
```

This gives a **quick and clean overview** of chat activity.

🔥 5.2 Sentiment Analysis

```
sentiment_df = helper.sentiment_analysis(selected_user, df)
ax.bar(sentiment_df['sentiment_label'], sentiment_df['message'])
```

Shows:

- Positive messages
- Negative messages
- Neutral messages

This is helpful for analyzing **emotional tone** of the chat.

5.3 Emoji Analysis

```
emoji_df = helper.emoji_helper(selected_user, df)
st.dataframe(emoji_df)
```

Also includes a **pie chart** of top emojis.

Shows:

- Most used emojis
- Emoji frequency distribution

Which helps understand **emotion & expressiveness**.

5.4 Monthly Timeline

```
timeline = helper.monthly_timeline(selected_user, df)
ax.plot(timeline['time'], timeline['message'])
```

Displays:

- Message frequency per month
 - Helps identify long-term activity trends
-

5.5 Daily Timeline

```
daily = helper.daily_timeline(selected_user, df)
ax.plot(daily['only_date'], daily['message'])
```

Shows:

- Daily messaging pattern
 - Useful for spotting **spikes** & **quiet days**
-

5.6 Activity Maps

Busy Day & Busy Month

```
busy_day = helper.week_activity_map(selected_user, df)
busy_month = helper.month_activity_map(selected_user, df)
```

Visualized using bar charts.

Weekly Heatmap

```
heatmap = helper.activity_heatmap(selected_user, df)
sns.heatmap(heatmap)
```

This heatmap shows:

- Activity by **weekday × hour**

- Helps identify "**peak usage times**"
-

5.7 Most Busy Users (Only When Overall Selected)

```
if selected_user == "Overall":  
    x, new_df = helper.most_busy_users(df)
```

Displays:

- A bar chart of **top active users**
- A table listing each user's message count

Very useful for **group chats**.

5.8 WordCloud

```
wc = helper.create_wordcloud(selected_user, df)  
ax.imshow(wc)
```

Shows frequently used words with:

- Bigger fonts for more frequent words
 - Gives instant insights into chat topics
-

5.9 Most Common Words

```
mc = helper.most_common_words(selected_user, df)  
ax.barh(mc[0], mc[1])
```

Displays a **horizontal bar chart** of the most used words.

Helps understand:

- Chat themes
 - Frequently discussed topics
 - Language style
-

★ 6. About Us Page

```
def about_us():  
    st.title("About Us - 24 Hour Version")
```

Provides:

- Creator information
 - Credits
 - Purpose of the app
-

★ 7. Sidebar Navigation

```
page = st.sidebar.radio("Go to", ["Home", "Analyze", "About Us"])
```

Allows navigation between:

- **Home Page**
- **Analyze Page**
- **About Us Page**

Clean and intuitive UI navigation.

8. Overall Workflow Summary

1. User uploads `.txt` chat in **24-hour format**
2. App preprocesses data using `preprocessors.py`
3. User selects participant or Overall
4. Presses **Show Analysis**
5. App renders:
 - Stats
 - Sentiment
 - Emoji analysis
 - Monthly & daily timelines
 - Activity maps
 - Heatmaps
 - WordCloud
 - Common words
 - Busy users (overall only)

This makes your app a **complete, user-friendly WhatsApp analytics tool**.

B. `preprocessors.py` Explanation(24-Hour Version)

```
import re
import pandas as pd

def preprocess(data):

    # Pattern that matches BOTH: 12/11/25, 21:55 - AND 12/11/2025, 21:55 -
    pattern = r'\d{1,2}/\d{1,2}/\d{2,4},\s\d{1,2}:\d{2}\s-\s'
```

```

messages = re.split(pattern, data)[1:]
dates = re.findall(pattern, data)

df = pd.DataFrame({'user_message': messages, 'message_date': dates})

# Try long year first, if it fails use short year
try:
    df['message_date'] = pd.to_datetime(df['message_date'],
                                       format='%d/%m/%Y, %H:%M - ')
except:
    df['message_date'] = pd.to_datetime(df['message_date'],
                                       format='%d/%m/%y, %H:%M - ')

df.rename(columns={'message_date': 'date'}, inplace=True)

# split user & message
users = []
msgs = []
for message in df['user_message']:
    entry = re.split(r'([\w\W]+?):\s', message)
    if entry[1:]:
        users.append(entry[1])
        msgs.append(" ".join(entry[2:]))
    else:
        users.append("group_notification")
        msgs.append(entry[0])

df["user"] = users
df["message"] = msgs
df.drop(columns=["user_message"], inplace=True)

# datetime breakdowns
df['only_date'] = df['date'].dt.date
df['year'] = df['date'].dt.year
df['month_num'] = df['date'].dt.month
df['month'] = df['date'].dt.month_name()

```

```

df['day'] = df['date'].dt.day
df['day_name'] = df['date'].dt.day_name()
df['hour'] = df['date'].dt.hour
df['minute'] = df['date'].dt.minute

# period generation
period = []
for hour in df['hour']:
    if hour == 23:
        period.append(f"{hour}-00")
    elif hour == 0:
        period.append(f"00-{hour+1}")
    else:
        period.append(f"{hour}-{hour+1}")

df['period'] = period

return df

```

This file is responsible for **parsing**, **cleaning**, and **structuring** WhatsApp chats that use **24-hour timestamps**.

It converts raw text into a fully-featured **Pandas DataFrame** that the analytics engine (`helper.py`) and UI (`apps.py`) can use.

✓ 1. Imports

```

import re
import pandas as pd

```

✓ Why these libraries?

Library	Purpose
<code>re</code>	Extract timestamps and split messages using regular expressions.

Library	Purpose
<code>pandas</code>	Build DataFrames, convert dates, and generate time-based features.

This file stays light and efficient—no heavy NLP here.

★ 2. Main Function: `preprocess(data)`

```
def preprocess(data):
```

- Accepts the **entire chat text** as one string.
- Returns a clean, analysis-ready **DataFrame**.

★ 3. Flexible Timestamp Pattern (Handles Both YY and YYYY)

```
pattern = r'\d{1,2}/\d{1,2}/\d{2,4},\s\d{1,2}:\d{2}\s-\s'
```

✓ This pattern matches timestamps like:

- `12/11/25, 21:55 -`
- `12/11/2025, 21:55 -`

✓ Why 2–4 digits for year?

Because WhatsApp exports sometimes use:

- 2-digit year → `25`
- 4-digit year → `2025`

So this regex handles **both formats automatically**.

★ 4. Extract Messages and Dates

```
messages = re.split(pattern, data)[1:]  
dates = re.findall(pattern, data)
```

✓ Explanation:

Operation	Result
<code>re.split()</code>	Splits the chat into individual message blocks.
<code>re.findall()</code>	Extracts all timestamps (same order as messages).
<code>[1:]</code>	Removes first empty item caused by split.

This gives **aligned lists**:

```
dates[0] → timestamp for messages[0]  
dates[1] → timestamp for messages[1]
```

Perfect for DataFrame construction.

★ 5. Create Initial DataFrame

```
df = pd.DataFrame({'user_message': messages, 'message_date': dates})
```

Columns so far:

Column	Meaning
user_message	raw sender + message text
message_date	raw timestamp

★ 6. Smart Timestamp Conversion (Handles 2 Formats)

```
try:
    df['message_date'] = pd.to_datetime(df['message_date'],
                                        format='%d/%m/%Y, %H:%M - ')
except:
    df['message_date'] = pd.to_datetime(df['message_date'],
                                        format='%d/%m/%y, %H:%M - ')
```

✓ Logic:

- Try parsing **YYYY year** first
- If it fails → parse **YY year**

You now support **both kinds of WhatsApp exports**.

Renaming for consistency:

```
df.rename(columns={'message_date': 'date'}, inplace=True)
```

★ 7. Split Username + Message Body

```
users = []
msgs = []
for message in df['user_message']:
    entry = re.split(r'([\w\W]+?):\s', message)
```

✓ WhatsApp message format:

Gaurav: Hello bro

But notifications like:

Messages to this group are now secured ...

have **no username**.

✓ Extraction logic:

```
if entry[1:]:
    users.append(entry[1])      # actual username
    msgs.append(" ".join(entry[2:])) # actual message text
else:
    users.append("group_notification")
    msgs.append(entry[0])
```

Result:

- Always separates **user** and **message** cleanly
- Classifies non-user lines as **group_notification**

★ 8. Update DataFrame with Cleaned Columns

```
df["user"] = users
df["message"] = msgs
df.drop(columns=["user_message"], inplace=True)
```

The DataFrame now has:

date	user	message
------	------	---------

Clean and efficient.

★ 9. Feature Engineering (Time-Based Columns)

```
df['only_date'] = df['date'].dt.date
df['year'] = df['date'].dt.year
df['month_num'] = df['date'].dt.month
df['month'] = df['date'].dt.month_name()
df['day'] = df['date'].dt.day
df['day_name'] = df['date'].dt.day_name()
df['hour'] = df['date'].dt.hour
df['minute'] = df['date'].dt.minute
```

✓ Why add these?

They power multiple charts:

Column	Used For
year	yearly activity
month, month_num	monthly timeline
day_name	busiest day analysis
hour	hourly heatmap
only_date	daily timeline
minute	detailed time analysis

These features transform raw data into **rich analytical insights**.

★ 10. Generate the Period Column

```
period = []
for hour in df['hour']:
    if hour == 23:
        period.append(f"{hour}-00")
    elif hour == 0:
        period.append(f"00-{hour+1}")
    else:
        period.append(f"{hour}-{hour+1}")
df['period'] = period
```

✓ Purpose:

Creates **hour ranges** like:

- 18-19
- 21-22
- 23-00

Used mainly in the **weekly activity heatmap**.

✓ Why important?

It groups messages into **time slots** instead of individual hours, improving heatmap clarity.

★ 11. Return Final Clean DataFrame

```
return df
```

Your output now contains:

Final Columns:

Column	Description
date	exact timestamp
user	sender
message	actual message text
only_date	date only
year	year extracted
month_num	month number
month	month name
day	day of month
day_name	weekday
hour	24-hour format
minute	minute
period	hourly interval

This DataFrame is fully compatible with:

- `apps.py` (24-hour GUI)
 - `helper.py` (analysis functions)
-

9. Flow Between `app.py` & `preprocessors.py`

1. User uploads `.txt` file in `app.py`.
2. File is sent to `preprocessors.preprocess()` → cleaned DataFrame.
3. `helper.py` functions use this DataFrame for:
 - **Stats** (`fetch_stats`)
 - **Timelines** (`monthly_timeline` , `daily_timeline`)
 - **Activity Maps** (`week_activity_map` , `month_activity_map` , `activity_heatmap`)
 - **Word analysis** (`create_wordcloud` , `most_common_words`)

- **Sentiment** (`sentiment_analysis`)

4. Streamlit displays **charts, tables, and word clouds** interactively.

✓ Key Takeaways for Notes

- `apps.py` → **UI + workflow controller**.
- `preprocessors.py` → **data cleaning + feature engineering**.
- `helper.py` → **analysis and visualization**.
- The structure allows **modular and scalable design**.
- The **12-hour version** is simpler but retains all core analysis: stats, timelines, activity, words, and sentiment.

Key Difference: 12-Hour vs 24-Hour WhatsApp Timestamps

`app.py -processor.py` VS `apps.py-processors.py`

1. WhatsApp Timestamp Formats

- **12-Hour Format (AM/PM)** – Typical in some locales (US, India optional):

12/05/23, 08:30 PM - User: Message text

- **24-Hour Format** – Used in other locales or custom export:

12/05/23, 20:30 - User: Message text

Notice:

Component	12-Hour	24-Hour
Hour	1-12 with AM/PM	0-23, no AM/PM
Example	08:30 PM	20:30

2. Regex Pattern Difference

12-Hour Processor:

```
pattern = r'\d{1,2}/\d{1,2}/\d{2,4},\s\d{1,2}:\d{2}\s[APMapm]{2}\s-\s'
```

- `\s[APMapm]{2}\s-\s` → looks for **AM/PM** after time.
- Matches **"08:30 PM - "** but **fails** for 24-hour time like 20:30.

24-Hour Processor:

```
pattern = '\d{1,2}/\d{1,2}/\d{2,4},\s\d{1,2}:\d{2}\s-\s'
```

- No AM/PM in regex.
- Matches **"20:30 - "** but fails if AM/PM is present.

3. DateTime Parsing Difference

12-Hour Processor:

```
df['message_date'] = pd.to_datetime(df['message_date'], format='%m/%d/%y, %I:%M %p - ')
```

- `%I` → 12-hour clock (1-12)
- `%p` → AM/PM
- Example: `08:30 PM` → `20:30` in 24-hour internally

24-Hour Processor:

```
df['message_date'] = pd.to_datetime(df['message_date'], format='%d/%m/%y, %H:%M - ')
```

- `%H` → 24-hour clock (0–23)
- No AM/PM needed
- Example: `20:30` → 20:30

4. Practical Impact

Feature	12-Hour Processor	24-Hour Processor
Regex pattern	Requires AM/PM in timestamp	Only numbers, no AM/PM
<code>pd.to_datetime</code>	Uses <code>%I:%M %p</code> format	Uses <code>%H:%M</code> format
Works with timestamps	"08:30 PM" or "07:15 AM"	"20:30" or "07:15"
Errors if wrong format	Yes, will fail to parse dates	Same, will fail if AM/PM exists
General usage	Mostly WhatsApp exports from phones with 12-hour clock	Exports with 24-hour clock

5. Other Parts of Preprocessor

Everything else (user splitting, period generation, daily/monthly features) **remains identical** in both processors. The **only change** is how **timestamps are read and interpreted**.

6. Summary

- **Main difference:** How the **timestamp string is parsed**.
- **12-hour processor:** handles AM/PM timestamps (`%I:%M %p`).
- **24-hour processor:** handles 24-hour timestamps (`%H:%M`).

- **Why it matters:** If you use the wrong processor for a file, the **dates won't parse correctly**, and all subsequent analysis (timelines, activity maps) will break.

C. `helper.py` Explanation

`helper.py` is **the analysis engine**. It takes the cleaned DataFrame from `preprocessors.py` and provides **statistics, timelines, word analysis, emoji analysis, and sentiment insights**. These results are then displayed in `app.py`.

Imports

```
from urlextract import URLExtract
from wordcloud import WordCloud
import pandas as pd
from collections import Counter
import emoji
import nltk
from nltk.sentiment.vader import SentimentIntensityAnalyzer
```

- **URLExtract** → Extracts URLs from text.
- **WordCloud** → Generates word cloud images.
- **pandas** → Data manipulation and aggregation.
- **Counter** → Counts word/emoji occurrences efficiently.
- **emoji** → Detects emojis in messages.
- **nltk + Vader** → Sentiment analysis (`SentimentIntensityAnalyzer`).

```
nltk.download('vader_lexicon')
extract = URLExtract()
```

```
sentiment_analyzer = SentimentIntensityAnalyzer()
```

- Downloads sentiment lexicon.
- Initializes URL extractor and sentiment analyzer.

1. **fetch_stats()** – Top-level Stats

```
def fetch_stats(selected_user,df):  
    if selected_user != 'Overall':  
        df = df[df['user'] == selected_user]  
  
    num_messages = df.shape[0] # Total messages  
  
    words = []  
    for message in df['message']:  
        words.extend(message.split()) # Total words  
  
    num_media_messages = df[df['message'] == '<Media omitted>\n'].shape  
    [0] # Media  
  
    links = []  
    for message in df['message']:  
        links.extend(extract.find_urls(message)) # URLs shared  
  
    return num_messages, len(words), num_media_messages, len(links)
```

- Filters messages by user or keeps all (**Overall**).
- Calculates:
 1. **Number of messages**
 2. **Number of words**
 3. **Media messages**

4. Links shared

- Used in `app.py` for the **Top Statistics** section.

2. `most_busy_users()` – Group-level User Activity

```
def most_busy_users(df):  
    x = df['user'].value_counts().head() # Top 5 active users  
    df = round((df['user'].value_counts() / df.shape[0]) * 100, 2).reset_index().r  
ename(  
    columns={'index': 'name', 'user': 'percent'})  
    return x, df
```

- **x** → bar chart of **top 5 users by message count**
- **df** → table showing % **contribution** of each user.
- Used when `selected_user == "Overall"` in `app.py`.

3. `create_wordcloud()` – Visualize Most Used Words

```
f = open('stop_hinglish.txt', 'r')  
stop_words = f.read()
```

- Reads a **stopword file** to ignore common words.
- Handles **Hinglish (Hindi + English)** words too.

```
temp['message'] = temp['message'].apply(remove_stop_words)  
df_wc = wc.generate(temp['message'].str.cat(sep=" "))
```

- Removes stopwords.
- Generates a **WordCloud** from remaining words.

- Displayed in `app.py` under **Wordcloud** section.
-

4. `most_common_words()` – Top Words

```
words = []
for message in temp['message']:
    for word in message.lower().split():
        if word not in stop_words:
            words.append(word)
most_common_df = pd.DataFrame(Counter(words).most_common(20))
```

- Collects all words, removes stopwords.
 - Counts top 20 words.
 - Returns a **DataFrame** for horizontal bar chart.
-

5. `emoji_helper()` – Emoji Analysis

```
emojis = [c for message in df['message'] for c in message if c in emoji.EMOJI_DATA]
emoji_df = pd.DataFrame(Counter(emojis).most_common(len(Counter(emojis))))
```

- Iterates through all messages and **extracts emojis**.
 - Counts occurrences.
 - Returns a DataFrame with **emoji and frequency**.
-

6. `monthly_timeline()` – Messages by Month

```
timeline = df.groupby(['year', 'month_num', 'month']).count()['message'].reset_index()
```

- Groups messages by **year and month**.
- Creates a **timeline column** like `"January-2023"`.
- Returns DataFrame for plotting **monthly trends**.

7. `daily_timeline()` – Messages by Day

```
daily_timeline = df.groupby('only_date').count()['message'].reset_index()
```

- Groups by **exact date**.
- Useful for **daily activity graphs**.

8. `week_activity_map()` – Messages by Weekday

```
return df['day_name'].value_counts()
```

- Counts messages for each day of the week.
- For bar chart **"Most Busy Day"** in `app.py`.

9. `month_activity_map()` – Messages by Month

```
return df['month'].value_counts()
```

- Counts messages for each month.
- For bar chart **"Most Busy Month"** in `app.py`.

10. `activity_heatmap()` – Hourly Heatmap

```
user_heatmap = df.pivot_table(index='day_name', columns='period', values='message', aggfunc='count').fillna(0)
```

- Creates a **matrix of day vs hour period**.
- Values = **number of messages**
- Displayed as **heatmap using seaborn**.

11. **sentiment_analysis()** – Sentiment Detection

```
df['sentiment'] = df['message'].apply(lambda x: sentiment_analyzer.polarity_scores(x))
df['sentiment_score'] = df['sentiment'].apply(lambda x: x['compound'])
df['sentiment_label'] = df['sentiment_score'].apply(lambda x: 'Positive' if x>0 else ('Negative' if x<0 else 'Neutral'))
sentiment_df = df.groupby('sentiment_label').count()['message'].reset_index()
```

- Uses **VADER Sentiment Analyzer** to calculate sentiment scores.
- Labels messages as: **Positive, Negative, Neutral**
- Groups counts by label for **bar chart** in `app.py`.

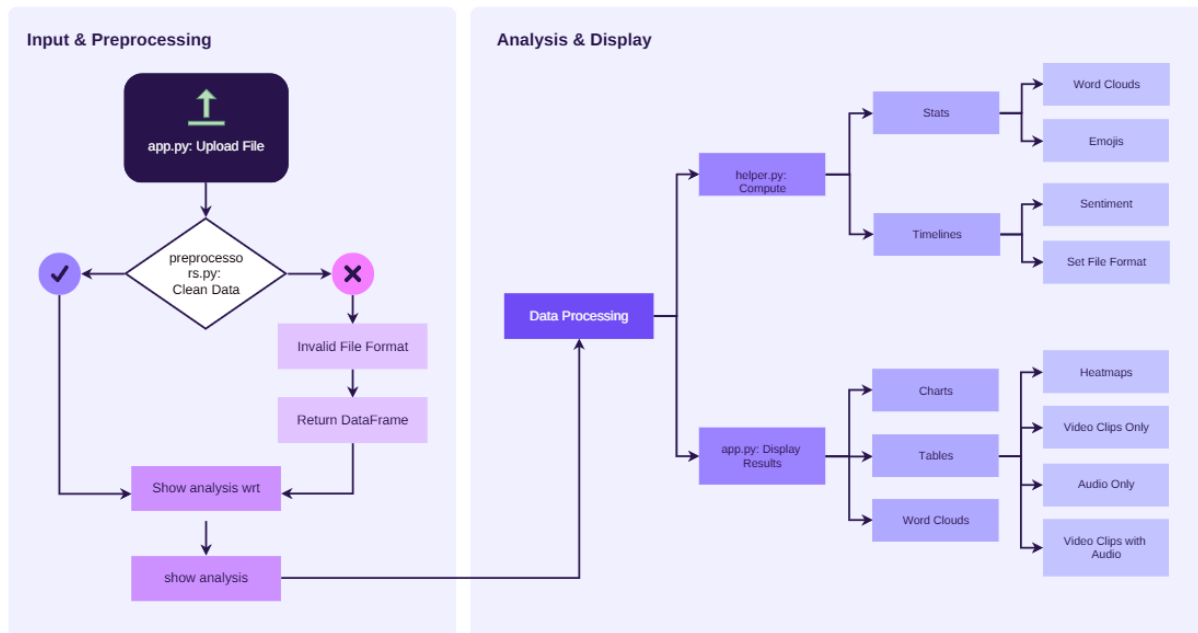
Summary Table of helper.py Functions

Function	Purpose	Output	Used in app.py Section
<code>fetch_stats</code>	Top statistics	num_messages, words, media, links	Stats area
<code>most_busy_users</code>	Top active users	Series & DataFrame	Most Busy Users
<code>create_wordcloud</code>	WordCloud of messages	WordCloud object	Wordcloud

Function	Purpose	Output	Used in app.py Section
<code>most_common_words</code>	Most frequent words	DataFrame	Most Common Words
<code>emoji_helper</code>	Count of emojis	DataFrame	Emoji Analysis
<code>monthly_timeline</code>	Monthly message trends	DataFrame	Monthly Timeline
<code>daily_timeline</code>	Daily message trends	DataFrame	Daily Timeline
<code>week_activity_map</code>	Messages per weekday	Series	Activity Map - Days
<code>month_activity_map</code>	Messages per month	Series	Activity Map - Months
<code>activity_heatmap</code>	Heatmap of hour/day	DataFrame	Weekly Activity Map
<code>sentiment_analysis</code>	Sentiment of messages	DataFrame	Sentiment Analysis

Flow in App

1. `app.py` → Upload file
2. `preprocessors.py` → Clean data → Return DataFrame
3. `helper.py` → Compute stats, timelines, word clouds, emojis, sentiment
4. `app.py` → Display charts, tables, word clouds, heatmaps



D.Stop_Hinglish.txt – Explanation

1. What it is

- This file is a **list of "stopwords"**.
- Stopwords are **common words that carry very little meaning** in text analysis.
- Your file contains a **mix of English and Hinglish words** (Hindi words written in Roman script, e.g., "bhai", "acha", "tum", "hai").
- It also contains:
 - Common English words: the, a, an, and, but, is
 - Common Hinglish conversational words: ka, kya, ho, hai, apna
 - Some filler/short forms: idk, lol, hmm

2. Why it is used

The main purpose of a stopwords file is to **improve text analysis** by ignoring words that **don't add meaningful information**.

Examples in your project:

1. WordCloud Generation

```
def create_wordcloud(selected_user, df):  
    f = open('stop_hinglish.txt', 'r')  
    stop_words = f.read()
```

- Words like `hai`, `ka`, `the`, `and` are **ignored**.
- This makes the WordCloud **focus on meaningful words**, e.g., `party`, `meeting`, `birthday`.

1. Most Common Words

```
for message in temp['message']:  
    for word in message.lower().split():  
        if word not in stop_words:  
            words.append(word)
```

- Removes stopwords so that **top 20 most frequent words are meaningful**.
- Without it, words like `hai`, `ka`, `ap`, `the` would dominate and **mislead analysis**.

3. Where it is linked

- In `helper.py`:
 - Functions:
 - `create_wordcloud()` → to generate cleaner word clouds
 - `most_common_words()` → to compute meaningful frequent words
- It is **loaded at runtime**:

```
f = open('stop_hinglish.txt', 'r')
stop_words = f.read()
```

- Stopwords are then **filtered from all user messages** in these functions.
- In **app.py**, this indirectly affects:
 - WordCloud section
 - Most Common Words section

4. Why Hinglish stopwords?

- WhatsApp chats often contain **Hindi words written in Roman script** (Hinglish).
- Without a Hinglish stopwords list:
 - Words like **hai** , **ka** , **tum** , **apka** would appear as most frequent.
 - This **reduces insight**, because we usually want **meaningful nouns or verbs**.

5. Summary for Notes

Aspect	Explanation
File name	<code>stop_hinglish.txt</code>
Type	Text file containing stopwords
Content	English stopwords + Hindi/Hinglish words + fillers
Purpose	Remove meaningless words for text analysis (WordCloud & most common words)
Linked in	<code>helper.py</code> → <code>create_wordcloud()</code> , <code>most_common_words()</code>
Benefit	Ensures visualizations and stats highlight meaningful content , not common fillers

✅ Tip for notes:

You can write:

"stop_hinglish.txt is a stopword list combining English and Romanized Hindi words. It is used in helper.py to remove common, low-value words from messages before generating WordClouds or calculating most frequent words, ensuring meaningful insights."

E.requirements.txt – Explanation

streamlit==1.30.0

matplotlib==3.8.1

seaborn==0.12.3

urlextract==1.5.0

wordcloud==1.8.2.2

pandas==2.1.0

emoji==2.6.0

nltk==3.8.1

Notes:

Streamlit → For the web app interface.

Matplotlib & Seaborn → For plots and heatmaps.

URLExtract → To extract URLs from chat messages.

WordCloud → To generate word clouds of chat messages.

Pandas → To handle and process the chat data.

Emoji → For emoji extraction and counting.

NLTK → For text processing and sentiment analysis.

4. Understanding the Heroku Deployment Files

✓ 1. Make Sure Your Project Has 3 Required Files

✓ Required files:

- Procfile
- requirements.txt
- setup.sh

✓ 2. Create Procfile (very important)

If your main file is **app.py**:

```
web: streamlit run app.py --server.port=$PORT --server.address=0.0.0.0
```

If your main file is **apps.py**:

```
web: streamlit run apps.py --server.port=$PORT --server.address=0.0.0.0
```

▲ No file extension. No .txt. Only: Procfile

★ HOW TO DEPLOY BOTH: **app.py** and **apps.py** (Two Different Heroku Apps)

Heroku deploys **ONE** app per repo.

So to deploy BOTH versions, you must deploy:

- App 1 → 12-hour version → **app.py**
- App 2 → 24-hour version → **apps.py**

I'll show you EXACT steps for each.

IMPORTANT: YOU NEED TWO DIFFERENT Procfiles

Heroku reads only **ONE** Procfile.

So for two deployments, you must:

- Keep same project
 - Deploy twice
 - Use two different Procfile versions
-

PART 1 — Deploy `app.py` (12-hour version)

Step 1 — Create `Procfile`

```
web: streamlit run app.py --server.port=$PORT --server.address=0.0.0.0
```

Step 2 — Push to Heroku

```
git init
git add .
git commit -m "Deploy 12-hour app"
heroku create whatsapp-analyzer-12hr
git push heroku main
```

Step 3 — Open

```
heroku open
```

Your **12-hour version** is now LIVE.

PART 2 — Deploy `apps.py` (24-hour version)

We will deploy it as a SECOND Heroku app.

Step 1 — Change the Procfile

Overwrite existing Procfile with:

```
web: streamlit run apps.py --server.port=$PORT --server.address=0.0.0.0
```

Step 2 — Commit changes

```
git add Procfile  
git commit -m "Switch to 24-hour version"
```

Step 3 — Create second Heroku app

```
heroku create whatsapp-analyzer-24hr
```

Step 4 — Push code

```
git push heroku main
```

Step 5 — Open the second app

```
heroku open
```

Your **24-hour version** is now LIVE.

3. Create **setup.sh**

Create a file:

```
setup.sh
```

Add:

```
mkdir -p ~/.streamlit/

echo "\
[general]\n\
email = \"your-email@gmail.com\"\n\
" > ~/.streamlit/credentials.toml

echo "\
[server]\n\
headless = true\n\
enableCORS = false\n\
port = $PORT\n\
" > ~/.streamlit/config.toml
```

Make it executable (Git Bash or WSL):

```
chmod +x setup.sh
```

✓ 4. Terminal: Initialize Git

```
git init  
git add .  
git commit -m "first deploy"
```

✓ 5. Login to Heroku

```
heroku login
```

✓ 6. Create Heroku App

```
heroku create whatsapp-chat-analyzer
```

Or let Heroku choose a name:

```
heroku create
```

✓ 7. Deploy to Heroku

If your branch is **main**:

```
git push heroku main
```

If your branch is **master**:

```
git push heroku master
```

✓ 8. Open Your App

```
heroku open
```

Done 🎉

Your Streamlit WhatsApp Analyzer is LIVE!

✓ Final Folder Structure (Heroku Ready)

```
WhatsApp-Chat-Analyzer
|
├── app.py
├── apps.py
├── setup.sh
├── Procfile
├── requirements.txt
├── helper.py
├── preprocessor.py
├── preprocessors.py
└── stop_hinglish.txt
```