Mood Prediction Based On Calendar Events

Contents

1.1 Section 1: Data Explanation

1.1.1 Overview of the Data

For this project, I have chosen to analyze my personal calendar events as the primary dataset. This dataset comprises entries from my Google Calendar, associated with my school email address, miray@uni.minerva.edu. The data spans from June 2021 to March 21, 2024, encompassing a wide array of events that mark my academic, personal, and social engagements during this period. In the dataset, each row shows an event with multiple features including its date, start time, end time, summary, number of attendees, status, etc.

1.1.2 Rationale for Data Selection

The choice of calendar events as my data source is motivated by the hypothesis that the nature and density of daily activities have a tangible impact on my mood. By analyzing this dataset, I aim to uncover patterns and insights that could predict my mood variations over and within each day, leveraging the rich contextual details embedded in each calendar entry. This project intends to bridge the quantitative and qualitative aspects of event scheduling (e.g., number of events, average length of events, the descriptions of events, etc.) with the qualitative experience of my mood fluctuations over different days and daily segments.

This dataset is particularly suitable for this analysis as it encapsulates a broad spectrum of my life's facets over a significant period. It includes academic deadlines, social engagements, personal tasks, and leisure activities, offering a holistic view of my routine and its potential effects on mood across different days and segments of a day. Ultimately, this project is about harnessing data to enhance my well-being and making informed decisions to predict and improve my daily feelings. I believe recognizing patterns that lead to negative or positive daily moods allows me to adjust my schedule strategically.

1.1.3 Data Acquisition

I obtained the calendar data using Google Takeout, a service by Google that allows users to export their data from the various Google services they use. I specifically requested an export of calendar events from my most active email account, which is predominantly used for school-related activities and personal management. This method comprehensively extracted my digital engagements over the specified period.

1.1.4 Data Sampling

I methodologically sampled this dataset to ensure a manageable yet representative selection of my daily engagements. Specifically, the dataset includes events from June 29, 2021, to March 21, 2024, with ~14 values removed due to the unavailability of event dates and times (important identification features). The extraction process further involved grouping events by date and 3 segments per date to focus my analysis only on days with at least one scheduled event. I chose this approach to facilitate the analysis of daily and daily-segmental moods based on the nature and volume of activities planned for each day and segment.

1.1.5 Ethical Considerations

In selecting this dataset, I have ensured that the data is personal but not private, adhering to the assignment's guidelines. The dataset does not include sensitive or confidential information and has been filtered to exclude entries that might inadvertently reveal personal details about third parties or sensitive personal information.

1.2 Section 2: Data Ingestion and Transformation

1.2.1 Setting Up the Environment

Before diving into the data transformation, the necessary Python packages are installed using pip. "icalendar" parses the iCalendar (.ics) files, a common format for calendar data. "pandas" is a powerful library for data manipulation and analysis, which will help in organizing the calendar data into a structured format.

```
[ ]: pip install icalendar pandas
```

```
Requirement already satisfied: icalendar in ./opt/anaconda3/lib/python3.8/site-
packages (5.0.11)
Requirement already satisfied: pandas in ./opt/anaconda3/lib/python3.8/site-
packages (2.0.3)
Requirement already satisfied: backports.zoneinfo in
./opt/anaconda3/lib/python3.8/site-packages (from icalendar) (0.2.1)
Requirement already satisfied: pytz in ./opt/anaconda3/lib/python3.8/site-
packages (from icalendar) (2021.1)
Requirement already satisfied: python-dateutil in
./opt/anaconda3/lib/python3.8/site-packages (from icalendar) (2.8.2)
Requirement already satisfied: tzdata>=2022.1 in
./opt/anaconda3/lib/python3.8/site-packages (from pandas) (2023.3)
Requirement already satisfied: numpy>=1.20.3 in
./opt/anaconda3/lib/python3.8/site-packages (from pandas) (1.24.3)
Requirement already satisfied: six>=1.5 in ./opt/anaconda3/lib/python3.8/site-
packages (from python-dateutil->icalendar) (1.15.0)
Note: you may need to restart the kernel to use updated packages.
```

1.2.2 Importing Libraries

The "icalendar" library is imported to handle the .ics file format, allowing us to parse the calendar data. "pandas" is used for data frame manipulation, making it easier to work with the data in

Python. The “datetime” module is imported to handle date and time information, which is crucial for processing calendar events.

```
[ ]: # Libraries

     from icalendar import Calendar, Event
     from datetime import datetime
     import pandas as pd
     import numpy as np
```

1.2.3 Extracting Event Properties

Here, I defined a function, “get_property_value,” to extract properties from each calendar event safely. Special handling is incorporated for date-related properties (dtstart, dtend, dtstamp, created, last-modified), converting them to pandas datetime objects for easier manipulation. Other properties are returned as strings. This ensures that the data is consistently formatted and null-safe.

```
[ ]: # Function to safely get property values

     def get_property_value(component, property_name):
         prop = component.get(property_name)
         if prop:
             if property_name in ['dtstart', 'dtend', 'dtstamp', 'created',␣
      ↪'last-modified']:
                 # Convert to datetime if the property is a date type
                 return pd.to_datetime(prop.dt)
             else:
                 # Return as string for all other types
                 return str(prop)
         return None
```

1.2.4 Loading and Parsing the Calendar Data

The calendar file is opened in binary read mode. The “Calendar.from_ical” method parses the .ics file, converting it into a format that can be iteratively processed to extract event details.

```
[ ]: # Load the ICS file

     with open('/Users/mirayozcan/Desktop/miray@uni.minerva.edu.ics', 'rb') as f: #␣
      ↪Use 'rb' for binary read mode
         gcal = Calendar.from_ical(f.read())
```

1.2.5 Extracting Event Details

A list, “events_data,” is initialized to hold the data for each event. The script iterates over components in the parsed calendar, focusing on those labeled “VEVENT” which denote individual events. For each event, relevant details are extracted using the “get_property_value” function and stored in a dictionary. These dictionaries are then appended to the “events_data” list, creating a collection of event records.

```
[ ]: # Create a list to hold all the event data
events_data = []

# Parse the events from the calendar
for component in gcal.walk():
    if component.name == "VEVENT":
        event = {
            'summary': get_property_value(component, 'summary'),
            'dtstart': get_property_value(component, 'dtstart'),
            'dtend': get_property_value(component, 'dtend'),
            'dtstamp': get_property_value(component, 'dtstamp'),
            'organizer': get_property_value(component, 'organizer'),
            'uid': get_property_value(component, 'uid'),
            'attendee': ";".join([str(a) for a in component.get('attendee',␣
 ↪[])]),
            'created': get_property_value(component, 'created'),
            'description': get_property_value(component, 'description'),
            'last-modified': get_property_value(component, 'last-modified'),
            'location': get_property_value(component, 'location'),
            'sequence': get_property_value(component, 'sequence'),
            'status': get_property_value(component, 'status'),
            'transp': get_property_value(component, 'transp')
        }
        events_data.append(event)
```

1.2.6 Transforming Data into a Pandas DataFrame

The list of event dictionaries is converted into a "pandas DataFrame," "df_events," providing a tabular representation of the calendar events. This format is well-suited for analysis and manipulation using "pandas" functionalities. The "head()" method displays the first few rows of the data frame, offering a preview of the structured data.

```
[ ]: # Convert the events data into a pandas DataFrame

df_events = pd.DataFrame(events_data)
df_events.head()
```

```
[ ]:                 summary                   dtstart  \
0   İsil and Miray Özcan 2021-08-25 07:15:00+00:00
1    Time Management Lab 2021-09-27 23:00:00+00:00
2  SS50 AND CS50 CLASSES 2021-10-04 16:00:00+00:00
3        Relaxation Time 2021-10-05 03:30:00+00:00
4                Wake Up 2021-10-05 14:45:00+00:00

                       dtend                   dtstamp  \
0  2021-08-25 07:30:00+00:00 2024-02-18 16:50:12+00:00
1  2021-09-28 00:00:00+00:00 2024-02-18 16:50:12+00:00
```

```
2  2021-10-04 19:30:00+00:00 2024-02-18 16:50:12+00:00
3  2021-10-05 04:30:00+00:00 2024-02-18 16:50:12+00:00
4  2021-10-05 15:00:00+00:00 2024-02-18 16:50:12+00:00

                   organizer                                  uid  \
0  mailto:miray@uni.minerva.edu  u27vltd63s8lihre9m8qgai0h8@google.com
1                       None  4igsh1itqeleukn7j13ci7lbq7@google.com
2                       None   11AC5951-5BF0-4981-8F43-1315CF7BCB60
3                       None   98AFF40D-AA40-42DC-9853-1BCECABEF32B
4                       None   58876813-C548-4619-804E-39F4281C16B5

                                            attendee  \
0  mailto:nisilozcan@gmail.com;mailto:miray@uni.m…
1
2
3
4

                 created  \
0 2021-08-23 15:55:07+00:00
1 2021-09-26 18:21:22+00:00
2 2021-09-27 23:41:59+00:00
3 2021-09-27 23:43:50+00:00
4 2021-09-27 23:45:06+00:00

                                         description  \
0  Event Name: 15 Minute Meeting\n\nNeed to make …
1    Do you always feel like you're running out o…
2                                               None
3                                               None
4                                               None

             last-modified location sequence     status  transp
0 2021-08-23 15:55:07+00:00     None     None  CONFIRMED  OPAQUE
1 2021-09-27 23:02:25+00:00     None     None  CONFIRMED  OPAQUE
2 2021-09-27 23:42:21+00:00     None        1  CONFIRMED  OPAQUE
3 2021-09-27 23:44:11+00:00     None        1  CONFIRMED  OPAQUE
4 2021-09-27 23:45:06+00:00     None        1  CONFIRMED  OPAQUE
```

1.3 Section 3: Data Cleaning, Pre-Processing, and Exploration (First and Second Pipelines)

1.3.1 1- Data Cleaning, Pre-Processing, and Exploration in the First Pipeline

1.3.2 Overview

The initial dataset, obtained from a .ics file and converted to .csv format for easier manipulation, required several cleaning and pre-processing steps to prepare it for analysis. The goal was to ensure data quality by removing inaccuracies and inconsistencies, such as missing values and duplicates, and to structure the dataset for effective analysis. Later, I utilized WPS office to perform data cleaning and pre-processing operations.

1.3.3 Cleaning Steps

1. **Removing Incomplete Events:** Initially, 14 rows with missing critical information such as date, duration, and start-end times were identified and removed. This step was crucial to ensure that the dataset only contained events that could be analyzed effectively.
2. **Eliminating Duplicates:** 26 duplicate entries, identified by identical summaries, dates, start, and end times, were removed. This step prevented skewed analysis results due to redundant data.

1.3.4 Pre-Processing Steps

1. **Creating Time-Related Columns:** New columns were created for the date, start time, end time, duration, and number of attendees for the events. These transformations were essential for later analysis, particularly for calculating event durations and scheduling patterns.
2. **Handling Missing Values:** After initial cleaning, further checks were performed to clean any missing values in the date, start time, end time, and duration columns, ensuring the dataset's completeness.
3. **Exclusion of Location Data:** Although the location data might have offered additional insights, it was excluded due to its sparse availability, focusing the analysis on time and participation metrics.
4. **Final Dataset Preparation:** The dataset was refined to include events until February 15, ensuring the analysis was current and relevant.

1.3.5 Feature Engineering Steps

With a cleaned and pre-processed dataset, several features were engineered to aid in the analysis:

1. **Date and Time Features:** The date column was converted to the datetime format to facilitate time-series analysis. Additional columns were created to capture the day of the month and the day of the week, providing insights into how event scheduling varies over time.
2. **Hourly Features:** The start and end times were parsed to extract the hour, allowing for the analysis of when events typically begin and end.

```
[ ]: # Read the cleaned and pre-processed CSV file
     df = pd.read_csv('/Users/mirayozcan/Desktop/events_cleaned_finalized.csv')

     df.head()
```

```
[ ]:    event_id       date start_time  end_time  duration  n_attendees  \
     0         1  8/25/2021    7:15:00   7:30:00        15          2.0
     1         2  9/27/2021   23:00:00   0:00:00        60          1.0
     2         3  10/4/2021   16:00:00  19:30:00       210          1.0
     3         4  10/5/2021    3:30:00   4:30:00        60          1.0
     4         5  10/5/2021   14:45:00  15:00:00        15          1.0
```

```
                                         description               summary
0  Event Name: 15 Minute Meeting\n\nNeed to make …  İsil and Miray Özcan
1  Do you always feel like you're running out of …    Time Management Lab
2                                                NaN  SS50 AND CS50 CLASSES
3                                                NaN        Relaxation Time
4                                                NaN                Wake Up
```

1.3.6 Aggregating Daily Summary

- **day_of_month** Represents the day of the month for each event, allowing analysis of activity patterns at a monthly granularity.
- **day_of_week:** Indicates the day of the week, with 1 for Monday and 7 for Sunday, to explore how schedules vary across the week.
- **n_events:** The total number of events per day, offering a sense of how packed each day is.
- **event_ids:** A concatenation of event IDs for the day, referring to individual events if needed.
- **total_duration:** The sum of durations for all events in a day (in minutes), indicating how much time scheduled activities occupy.
- **avg_duration:** The average duration of events on that day (in minutes), providing a measure of typical event length.
- **start_first_event:** The hour at which the day's first event starts, helping to understand how early activities begin.
- **end_last_event:** The hour the last event of the day ends, indicating how late activities are scheduled.
- **longest_event:** The duration of the longest event of the day (in minutes), highlighting the most significant time commitment.
- **total_attendees:** The sum of attendees for all events, which could correlate with the day's social intensity.
- **avg_attendees:** The average number of attendees per event, offering insight into the typical event size.

[ ]:
```python
import pandas as pd

# Convert 'date' column to datetime format for easier manipulation
df['date'] = pd.to_datetime(df['date'])

# Extract and create features related to the timing of events
df['day_of_month'] = df['date'].dt.day  # Day of the month for each event
df['day_of_week'] = df['date'].dt.dayofweek + 1  # Day of the week, adjusted to␣
 ↪make Monday=1

# Extract start and end hours from the time strings, ensuring to handle␣
 ↪non-numeric characters correctly
df['start_hour'] = df['start_time'].str.split(':').str[0].astype(int)  # Hour␣
 ↪the event starts
df['end_hour'] = df['end_time'].str.split(':').str[0].astype(int)  # Hour the␣
 ↪event ends
```

```
# Aggregate data by date to create a daily summary
daily_summary = df.groupby('date').agg(
    day_of_month=('day_of_month', 'first'),  # Day of the month for each␣
  ↪aggregated day
    day_of_week=('day_of_week', 'first'),  # Day of the week for each␣
  ↪aggregated day
    n_events=('date', 'size'),  # Total number of events per day
    event_ids=('event_id', lambda x: ', '.join(x.astype(str))),  # Concatenated␣
  ↪IDs of events
    total_duration=('duration', 'sum'),  # Total duration of all events in a day
    avg_duration=('duration', 'mean'),  # Average duration of events per day
    start_first_event=('start_hour', 'min'),  # Start hour of the first event
    end_last_event=('end_hour', 'max'),  # End hour of the last event
    longest_event=('duration', 'max'),  # Duration of the longest event
    total_attendees=('n_attendees', 'sum'),  # Total number of attendees in all␣
  ↪events
    avg_attendees=('n_attendees', 'mean')  # Average number of attendees per␣
  ↪event
).reset_index()

# Sort the daily summary by date for chronological analysis
daily_summary_sorted = daily_summary.sort_values(by='date', ascending=True).
  ↪reset_index(drop=True)

# Display the first few rows of the daily summary to verify the structure
daily_summary_sorted.head()
```

```
[ ]:         date  day_of_month  day_of_week  n_events    event_ids  \
     0 2021-06-29            29            2         1          324
     1 2021-07-23            23            5         1          309
     2 2021-08-12            12            4         2     295, 310
     3 2021-08-23            23            1         1          291
     4 2021-08-25            25            3         3  1, 299, 300

        total_duration  avg_duration  start_first_event  end_last_event  \
     0              15     15.000000                 20              20
     1              15     15.000000                 21              21
     2             120     60.000000                  4               5
     3              60     60.000000                  0               1
     4             265     88.333333                  0               7

        longest_event  total_attendees  avg_attendees
     0             15              2.0       2.000000
     1             15              2.0       2.000000
     2             60              4.0       2.000000
     3             60              1.0       1.000000
     4            160              4.0       1.333333
```

This section meticulously processed and transformed the event data into a daily summary format, ready for further analysis. The aggregated data encapsulates both quantitative metrics (e.g., number of events, total and average durations) and qualitative aspects (e.g., start of the first event, end of the last event), providing a comprehensive overview of each day's schedule. These features are pivotal for analyzing how different types of days might influence mood, forming the basis for the predictive modeling to follow. This step not only refines the dataset for machine learning but also aligns it with the project's goal of understanding and potentially improving daily mood through schedule adjustments.

1.3.7 Labeling the Daily Summary Dataset

Labeling the dataset is a critical step in preparing for supervised learning tasks. In this project, the goal was to predict the daily mood based on calendar events. To achieve this, I labeled each day in the dataset with a mood category based on the nature and scheduling of events. The mood categories defined were **Busy**, **Productive**, **Relaxed**, and **Overwhelmed**.

The labeling process involved a detailed review of the sorted daily summaries, considering quantitative data (e.g., number of events, total duration) and qualitative aspects (e.g., types of events, spacing between events). Additionally, personal recollection of each day's overall mood and impact was considered to assign the most fitting label. The criteria for each mood category were as follows: 1. **Busy:** Days marked by a high number of events and a significant total duration, often with minimal gaps between events, indicating a tightly packed schedule. **Indicator:** High number of events and total duration, minimal rest periods.

2. **Productive:** Characterized by days with a moderate number of events, focusing on meaningful activities that contribute towards personal or professional goals. **Indicator:** Moderate number of significant events, focus on goal-oriented activities.

3. **Relaxed:** Days with fewer events, often involving leisure or personal care activities, and characterized by ample downtime. **Indicator:** Few events, significant gaps between events, presence of leisure activities.

4. **Overwhelmed:** Days that feel excessively demanding, potentially due to the nature of the events rather than just the schedule density. These might include high-stakes meetings or deadlines. **Indicator:** High-stress events, possibly fewer but longer and more demanding activities, ending late.

I conducted the labeling manually to incorporate a nuanced understanding and personal memory of each day's impact on mood. This approach ensured that the labels reflected a comprehensive assessment of each day's schedule and its emotional impact.

```
[ ]: # Exporting the data for manual labeling
     daily_summary_sorted.to_csv('/Users/mirayozcan/Desktop/daily_summary_sorted.
      ↪csv', index=False)
```

After carefully reviewing each day's summary and considering both the structured data and personal recall, I assigned labels to each row in the dataset. This process required a thoughtful balance between the objective data provided by the calendar events and the subjective experience of each day.

After labeling the dataset, I imported it back into Python for further analysis. This step marked the transition from data preparation to the application of machine learning models.

```
[ ]: # Re-importing the labeled dataset
     df_labeled = pd.read_csv('/Users/mirayozcan/Desktop/
      ↪daily_summary_sorted_labeled.csv')

     df_labeled.head()
```

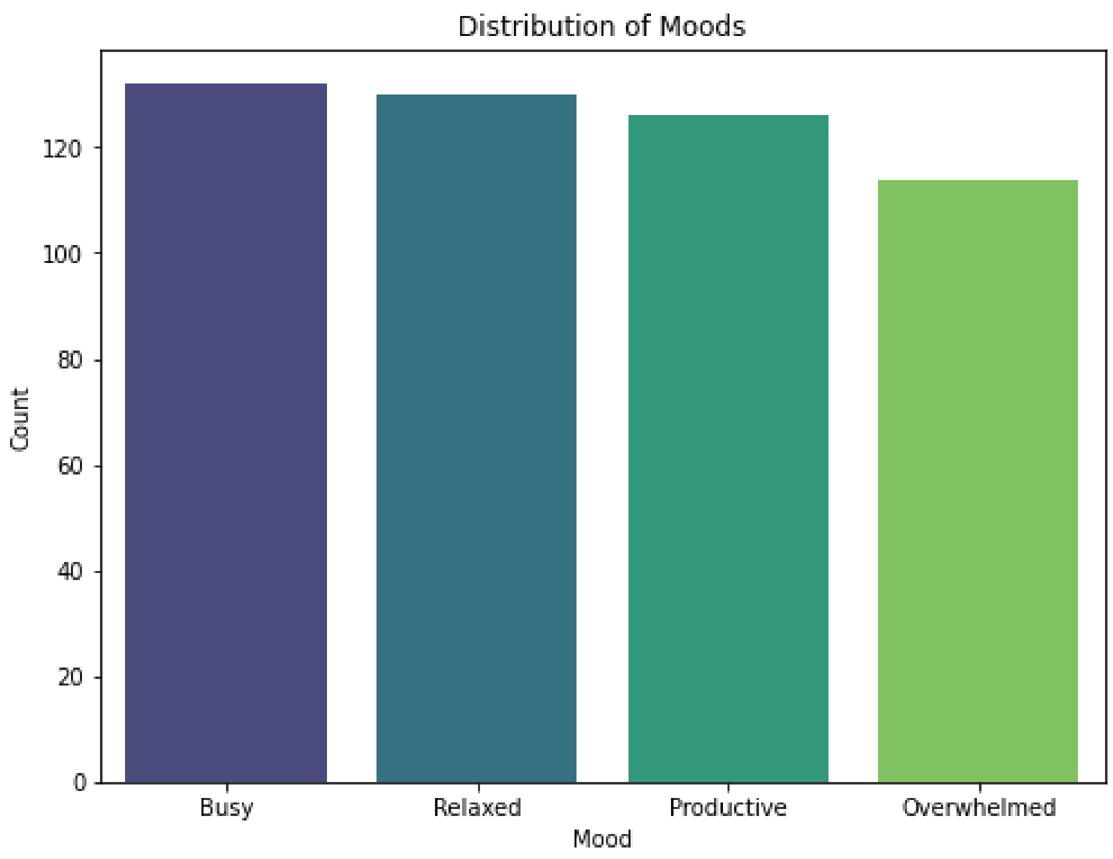
```
[ ]:         date  day_m  day_w  n_events event_ids  sum_duration  avg_duration  \
     0  6/29/2021     29      2         1       289            15          15.0
     1  7/23/2021     23      5         1       275            15          15.0
     2  8/12/2021     12      4         2  262, 276           120          60.0
     3  8/23/2021     23      1         1       258            60          60.0
     4  8/25/2021     25      3         2    1, 266           265         132.5

        start_first_event  end_last_event  longest_event  sum_attendees  \
     0                 20              20             15              2
     1                 21              21             15              2
     2                  4               5             60              4
     3                  0               1             60              1
     4                  7               7            250              3

        avg_attendees         mood
     0            2.0      Relaxed
     1            2.0      Relaxed
     2            2.0  Overwhelmed
     3            1.0      Relaxed
     4            1.5      Relaxed
```

1.3.8 Exploratory Analysis on the First Pipeline Data

```
[ ]: # Visualization Libraries

     import matplotlib.pyplot as plt
     import seaborn as sns
```

Bar Chart: Distribution of Moods

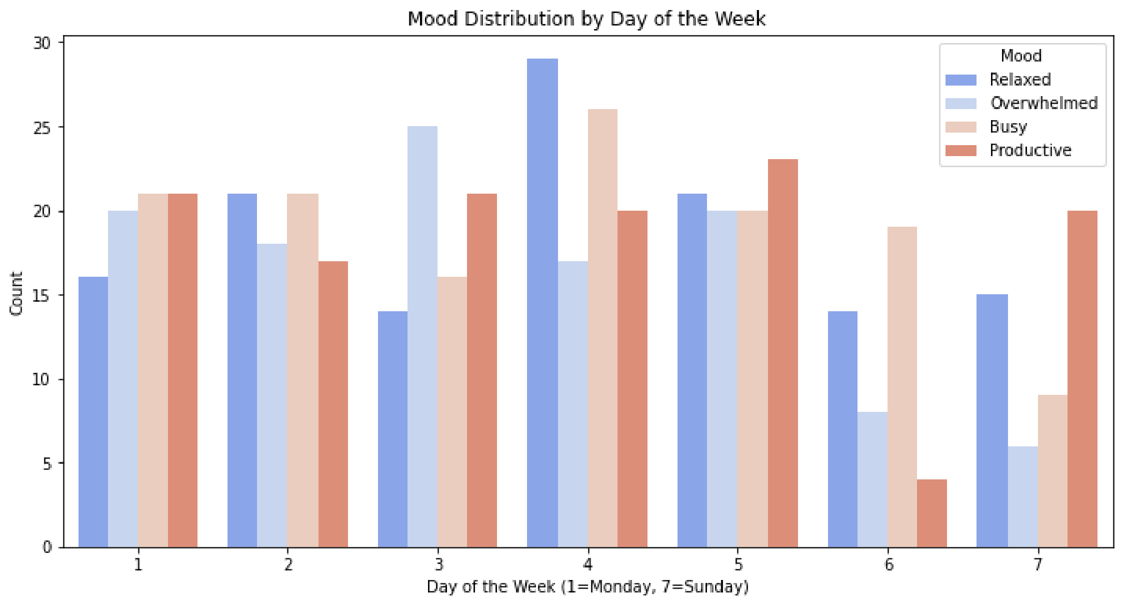
```
[ ]: # Mood distribution
     mood_counts = df_labeled['mood'].value_counts()
     plt.figure(figsize=(8, 6))
     sns.barplot(x=mood_counts.index, y=mood_counts.values, palette="viridis")
     plt.title('Distribution of Moods')
     plt.xlabel('Mood')
     plt.ylabel('Count')
     plt.show()
```



The bar chart represents the frequency of each mood category within the dataset. The length of each bar corresponds to the count of days categorized by the respective mood. This visualization is crucial as it illustrates the balance of mood occurrences over the analyzed period, which is a factor that will reduce the bias in our conclusions after training the model and looking at the results.

Mood Distribution by Day of the Week

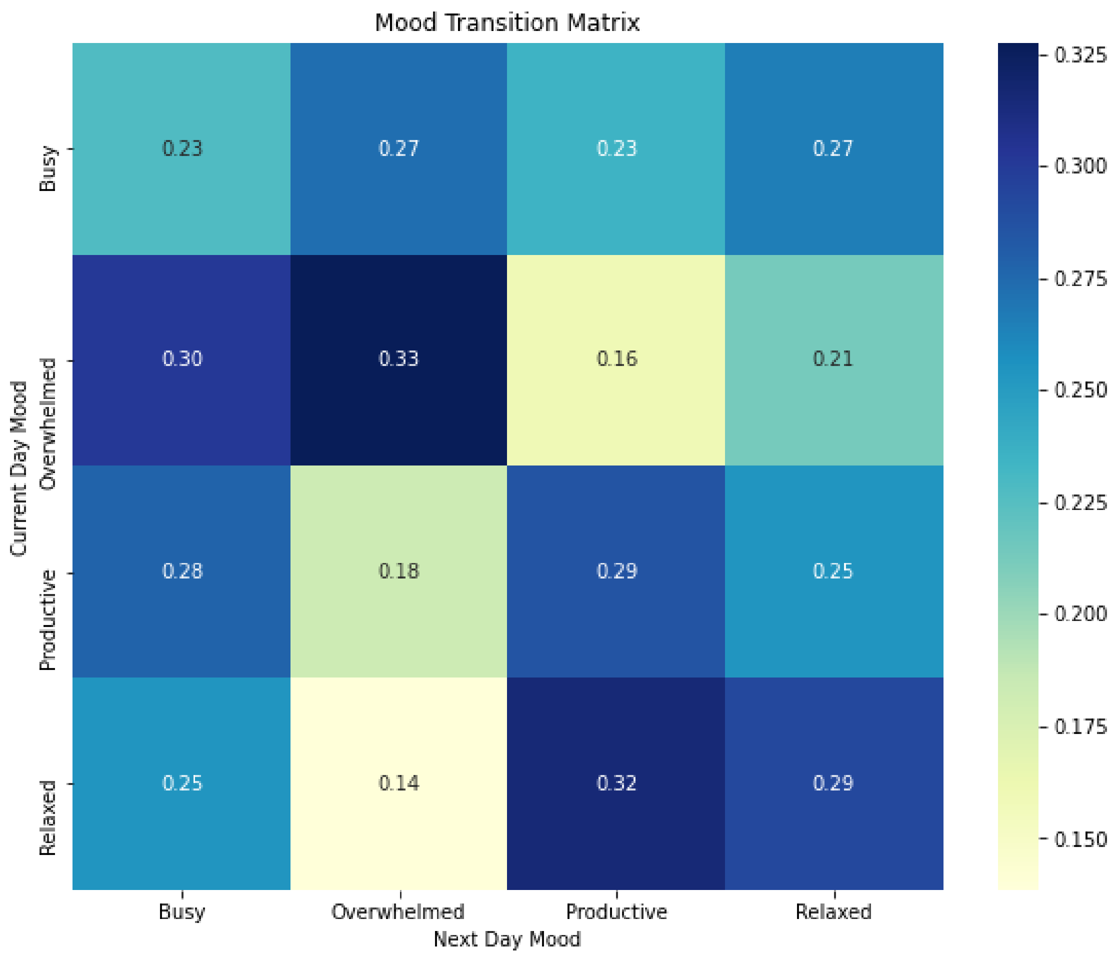
[ ]:
```
plt.figure(figsize=(12, 6))
sns.countplot(x='day_w', hue='mood', data=df_labeled, palette="coolwarm")
plt.title('Mood Distribution by Day of the Week')
plt.xlabel('Day of the Week (1=Monday, 7=Sunday)')
plt.ylabel('Count')
plt.legend(title='Mood')
plt.show()
```



The distribution of moods varies across the week, indicating a potential pattern or trend in how different moods are experienced on specific days. Some moods appear more frequently on certain days. For instance, the 'Productive' mood peaks mid-week, which correlates with the traditional work or school week's flow, where individuals gain momentum post-Monday. There is a noticeable change in mood frequencies on weekends (days 6 and 7), where 'Relaxed' moods may increase due to typical social norms of weekends being days of rest or leisure. The 'Overwhelmed' mood is significant around the middle of the week, which could be associated with the culmination of workload or stress. The 'Busy' mood is consistent throughout the week with slight variations, suggesting a relatively stable activity level across different days. This mood distribution can inform personal or organizational scheduling by highlighting days where there might be a tendency to feel more overwhelmed or productive, allowing for better planning to manage stress and enhance productivity.

Mood Transition Matrix

```
[ ]: df_labeled['next_mood'] = df_labeled['mood'].shift(-1)
     mood_transition_matrix = pd.crosstab(df_labeled['mood'],␣
      ↪df_labeled['next_mood'], normalize='index')

     plt.figure(figsize=(10, 8))
     sns.heatmap(mood_transition_matrix, annot=True, cmap="YlGnBu", fmt=".2f")
     plt.title('Mood Transition Matrix')
     plt.xlabel('Next Day Mood')
     plt.ylabel('Current Day Mood')
     plt.show()
```



The diagonal elements of the matrix show the probabilities of staying in the same mood from one day to the next. For instance, days categorized as ‘Productive’ have a 29% chance of being followed by another ‘Productive’ day. The matrix reveals patterns in mood shifts. For example, ‘Overwhelmed’ days tend to be followed by ‘Productive’ days with a probability of 33%, which could indicate a recovery or response mechanism to a previously stressful day. Certain transitions, such as from ‘Relaxed’ to ‘Overwhelmed,’ are less likely to occur, suggesting that days off or leisurely days do not commonly lead directly to overwhelming days. Some transitions appear symmetrical, such as from ‘Busy’ to ‘Relaxed’ and vice versa, both with a probability of around 25%. This could suggest a balance in the pace of activities from one day to the next.

The rest of the visualizations from the first pipeline were skipped in this report, since they will be reused with updated data for the second pipeline.

1.3.9 Data Cleaning, Preprocessing, and Exploration in the Second Pipeline

1.3.10 Overview of the Updates

In the first pipeline, one significant challenge was the limited data availability, which restricted the depth of analysis and model performance. With initial sentiments spread thinly across classes, the models struggled to distinguish between nuanced moods across four categories due to insufficient examples per class. Addressing this, the second pipeline emphasized data augmentation to bolster the dataset and refine predictive capabilities. Techniques such as subdividing events and days into smaller samples and employing generative AI for paraphrasing event descriptions were introduced to expand the dataset artificially. This approach aimed to amplify the data to approach the recommended threshold of 10^3 samples per class.

Additionally, based on the insightful feedback, the complexity of mood categories was reduced from 4 categories to a binary classification of "positive" and "negative," simplifying the target and focusing on a more robust signal in the data. Daily segments with events and non-events were treated as distinct input categories, revealing mood patterns tied to the presence or absence of scheduled activities. Days were dissected into smaller segments (3 daily segments), considering the "no event" periods as meaningful predictors of mood. This granular view allowed the inclusion of 'nothing' as a feature, providing a more holistic representation of daily experiences. This nuanced encoding helped to capture the entire spectrum of daily and segment-specific dynamics, including the influence of free time on mood.

Finally, the first pipeline only considered the numeric features of the dataset, losing valuable information that could significantly impact the mood predictions coming from qualitative features, such as event summaries and descriptions. In this pipeline, additional preprocessing, feature engineering, and cleaning steps were also followed to include textual data alongside numeric data to train a wide range of machine learning models that could give a more extended view of the patterns in the dataset.

1.3.11 Pre-Processing Step 1: Modifying the Events and Divide Events into Parts If Needed

The pre-processing of event data in the second pipeline has been designed to account for the daily flow of activities and their potential impact on mood. Recognizing that certain events may span across different segments of the day, it was imperative to modify the representation of these events in the dataset. This section outlines the process undertaken to chop up events into parts, ensuring each part falls within a designated daily segment, thereby refining the data for a segment-based aggregation.

```
[ ]: # Libraries

     from itertools import product
```

```
[ ]: # Read the cleaned and pre-processed CSV file

     df = pd.read_csv('/Users/mirayozcan/Desktop/events_cleaned_finalized.csv')
     df.head()
```

```
[ ]:    event_id        date start_time  end_time  duration  n_attendees  \
     0         1   8/25/2021    7:15:00   7:30:00        15          2.0
     1         2   9/27/2021   23:00:00   0:00:00        60          1.0
     2         3   10/4/2021   16:00:00  19:30:00       210          1.0
     3         4   10/5/2021    3:30:00   4:30:00        60          1.0
     4         5   10/5/2021   14:45:00  15:00:00        15          1.0
```

```
                                         description               summary
0  Event Name: 15 Minute Meeting\n\nNeed to make …   İsil and Miray Özcan
1  Do you always feel like you're running out of …    Time Management Lab
2                                                NaN  SS50 AND CS50 CLASSES
3                                                NaN        Relaxation Time
4                                                NaN                Wake Up
```

In the initial dataset, events that were on the cusp of daily segments—defined by the start and end times including 00:00:00, 08:00:00, or 16:00:00—were manually adjusted to align perfectly within these segments. This manual preprocessing maintained the integrity of each event's core attributes (date, number of attendees, description, and summary) while adapting their timing to fit within the morning, daytime, or evening segments.

```
[ ]: # Read the manually chopped events

     df_chopped = pd.read_csv('/Users/mirayozcan/Desktop/events_chopped.csv')
     print('The number of rows in this dataset is:', len(df_chopped))
     df_chopped.head()
```

```
The number of rows in this dataset is: 1701
```

```
[ ]:   event_id       date start_time  end_time  duration  n_attendees  \
    0         1  8/25/2021    7:15:00   7:30:00        15          2.0
    1         2  9/27/2021   23:00:00   0:00:00        60          1.0
    2         3  10/4/2021   16:00:00  19:30:00       210          1.0
    3         4  10/5/2021    3:30:00   4:30:00        60          1.0
    4         5  10/5/2021   14:45:00  15:00:00        15          1.0

                                         description               summary  \
    0  Join İsil and Miray Özcan for an interactive w…   İsil and Miray Özcan
    1  Participate in a comprehensive Time Management…    Time Management Lab
    2  Engage in both Social Science 50 and Computer …  SS50 AND CS50 CLASSES
    3  Indulge in a relaxation session to unwind and …        Relaxation Time
    4  Start your day energetically with a morning wa…                Wake Up

       is_within_segment  segment  start_hour  end_hour
    0               True  morning           7         7
    1               True  evening          23         0
    2               True  evening          16        19
    3               True  morning           3         4
    4               True  daytime          14        15
```

A critical verification step ensured that no events spanned across different segments post-manual processing. A function was created to check if an event's start and end times fell within the defined morning, daytime, or evening boundaries. The function returned a Boolean value for each event, verifying its correct placement within a segment.

```
[ ]: # Verify that no events span across different day segments after manual␣
     ↪preprocessing

     def check_event_segment(row):
         # Adjust end time for comparison
         end_time = row['end_time'] if row['end_time'] != '00:00:00' else '24:00:00'

         # Convert start and end times to datetime objects for easier comparison
         start_dt = datetime.strptime(row['start_time'], '%H:%M:%S')
         end_dt = datetime.strptime(end_time, '%H:%M:%S')

         # Convert times to minutes since midnight
         start_min = start_dt.hour * 60 + start_dt.minute
         end_min = end_dt.hour * 60 + end_dt.minute

         # Define segment boundaries
         segments = {'morning': (0, 8 * 60), 'daytime': (8 * 60, 16 * 60), 'evening':
      ↪ (16 * 60, 24 * 60)}

         for segment, (start_bound, end_bound) in segments.items():
             if start_min >= start_bound and end_min <= end_bound:
                 return True
         return False

     # Apply the updated function
     df_chopped['is_within_segment'] = df_chopped.apply(check_event_segment, axis=1)

     # Identify events that span across different segments
     events_outside_segments = df_chopped[df_chopped['is_within_segment'] == False]

     # Display these events to review
     if len(events_outside_segments) > 0:
         events_outside_segments
     else:
         print("All events are modified according to day segments appropriately.")
```

```
All events are modified according to day segments appropriately.
```

To facilitate segment-based analysis, each event was labeled with the time of day it occurred—morning, daytime, or evening. A labeling function mapped the start time of each event to the corresponding segment. This was primarily for the purpose of assigning events to daily segments while refining the data for a segment-based aggregation.

```
[ ]: # Label each event with its day segment

     def label_event_segment(start_time):
         hour = int(start_time.split(':')[0])
```

```
    if 0 <= hour < 8:
        return 'morning'
    elif 8 <= hour < 16:
        return 'daytime'
    else:  # 16 <= hour < 24
        return 'evening'

# Apply the labeling function to the start_time column to create the segment␣
 ↪labels
df_chopped['segment'] = df_chopped['start_time'].apply(label_event_segment)
print('The number of rows in this dataset is;', len(df_chopped))
df_chopped.head()
```

```
The number of rows in this dataset is; 1701
```

[ ]:
```
   event_id       date start_time  end_time  duration  n_attendees  \
0         1  8/25/2021    7:15:00   7:30:00        15          2.0
1         2  9/27/2021   23:00:00   0:00:00        60          1.0
2         3  10/4/2021   16:00:00  19:30:00       210          1.0
3         4  10/5/2021    3:30:00   4:30:00        60          1.0
4         5  10/5/2021   14:45:00  15:00:00        15          1.0

                                         description              summary  \
0  Join İsil and Miray Özcan for an interactive w…  İsil and Miray Özcan
1  Participate in a comprehensive Time Management…   Time Management Lab
2  Engage in both Social Science 50 and Computer …  SS50 AND CS50 CLASSES
3  Indulge in a relaxation session to unwind and …        Relaxation Time
4  Start your day energetically with a morning wa…                Wake Up

   is_within_segment  segment  start_hour  end_hour
0               True  morning           7         7
1               True  evening          23         0
2               True  evening          16        19
3               True  morning           3         4
4               True  daytime          14        15
```

The segmentation of events laid the groundwork for segment-based daily aggregations, enabling a more granular analysis of daily patterns. With each event neatly fitting into a morning, daytime, or evening category, the dataset now provided a more refined foundation for exploring how different parts of the day impact mood.

[ ]:
```
df_chopped.to_csv('/Users/mirayozcan/Desktop/events_chopped.csv', index=False)
```

1.3.12 Pre-Processing Step 2: Creating Daily Segment-Based Aggregation (Each Date Is Represented By 3 Parts)

For the second pre-processing step, the focus was on creating a daily segment-based aggregation of events. The intention behind this methodical process was to enrich the dataset by presenting each day in three instances, corresponding to morning, daytime, and evening segments. This expanded view facilitated a tri-fold increase in the dataset size and a more nuanced exploration of daily mood variations.

First, a comprehensive DataFrame of all possible date and segment combinations was established. By pairing each unique date with the three time segments of the day, a template was formed to accommodate the aggregation of events.

```
[ ]: # Generate a DataFrame of all date and segment combinations

unique_dates = df_chopped['date'].drop_duplicates()
segments = ['morning', 'daytime', 'evening']
date_segment_combinations = pd.DataFrame(list(product(unique_dates, segments)),␣
 ↪columns=['date', 'segment'])
```

The ‘start_time’ and ‘end_time’ columns were condensed to their hour component to align events with their respective segments more effectively. This transformation was pivotal for the subsequent aggregation step.

```
[ ]: # Convert 'start_time' and 'end_time' to just the hour part before aggregation

df_chopped['start_hour'] = pd.to_datetime(df_chopped['start_time'], format='%H:
 ↪%M:%S').dt.hour
df_chopped['end_hour'] = pd.to_datetime(df_chopped['end_time'],␣
 ↪errors='coerce', format='%H:%M:%S').dt.hour
df_chopped['end_hour'] = df_chopped['end_hour'].fillna(24)  # Assuming '00:00:
 ↪00' as end_time is considered as 24
```

With these transformations in place, events were aggregated based on the date and segment, creating a structured summary of activities within each daily segment.

```
[ ]: # Aggregate existing events by date and segment with the adjusted hour␣
 ↪extraction

aggregated_events = df_chopped.groupby(['date', 'segment']).agg({
    'event_id': lambda x: ', '.join(map(str, x)),
    'duration': ['sum', 'mean'],
    'n_attendees': ['size', 'sum', 'mean'],
    'start_hour': 'min',
    'end_hour': 'max',
}).reset_index()

# Adjust column naming to reflect the inclusion of start and end hours
aggregated_events.columns = ['date', 'segment', 'event_ids', 'total_duration',␣
 ↪'avg_duration',
                             'n_events', 'total_attendees', 'avg_attendees',
                             'start_first_event', 'end_last_event']
```

Following aggregation, the newly formed segment-based event summaries were merged back into the complete list of date and segment combinations. Missing values were addressed by assigning placeholders, ensuring consistency and maintaining data integrity.

[ ]:
```python
# Proceed with merging, filling missing values, and adding day_of_month,␣
↪day_of_week as before

daily_summary_chopped = pd.merge(date_segment_combinations, aggregated_events,␣
↪on=['date', 'segment'], how='left')

daily_summary_chopped.fillna({
    'event_ids': 0, # numeric and distinguishable placeholder for missing values
    'total_duration': 0,
    'avg_duration': 0,
    'n_events': 0,
    'total_attendees': 0,
    'avg_attendees': 0,
    'start_first_event': -1, # numeric and distinguishable placeholder for␣
↪missing values
    'end_last_event': -1 # same as above
}, inplace=True)

# Ensure 'date' column is of datetime type after merging and before extracting␣
↪components
daily_summary_chopped['date'] = pd.to_datetime(daily_summary_chopped['date'])
```

Post-merging, the 'date' column underwent conversion to the datetime format to facilitate the extraction of 'day_of_month' and 'day_of_week', essential features for further analysis.

[ ]:
```python
# Extract 'day_of_month' and 'day_of_week'

daily_summary_chopped['day_of_month'] = daily_summary_chopped['date'].dt.day
daily_summary_chopped['day_of_week'] = daily_summary_chopped['date'].dt.
↪dayofweek + 1

# Rest of the preprocessing

daily_summary_chopped['segment'] = pd.
↪Categorical(daily_summary_chopped['segment'], categories=['morning',␣
↪'daytime', 'evening'], ordered=True)
daily_summary_chopped.sort_values(by=['date', 'segment'], inplace=True)
daily_summary_chopped.reset_index(drop=True, inplace=True)
```

A new column 'has_event' was introduced to denote the presence or absence of events within each segment, while categorical values within the 'segment' column were encoded into numeric representations to prepare the data for machine learning algorithms.

[ ]:
```python
# Add a new column 'has_event' that is 1 if there's at least one event in the␣
↪segment and 0 otherwise
```

```
daily_summary_chopped['has_event'] = (daily_summary_chopped['n_events'] > 0).
↪astype(int)
daily_summary_chopped[['date', 'segment', 'event_ids', 'has_event']].head()
```

```
[ ]:         date  segment event_ids  has_event
     0 2021-06-29  morning         0          0
     1 2021-06-29  daytime         0          0
     2 2021-06-29  evening       325          1
     3 2021-07-23  morning         0          0
     4 2021-07-23  daytime         0          0
```

```
[ ]: # Encode categorical variables like "morning," "daytime," and "evening" into␣
     ↪numeric values
     # as a common practice in preparing data for machine learning algorithms.

     # Map segments to ordinal values: morning=0, daytime=1, evening=2

     segment_mapping = {
         'morning': 0,
         'daytime': 1,
         'evening': 2
     }

     # Apply the mapping to the 'segment' column
     daily_summary_chopped['segment_numeric'] = daily_summary_chopped['segment'].
     ↪map(segment_mapping)
     daily_summary_chopped[['date', 'segment', 'segment_numeric']].head()
```

```
[ ]:         date  segment segment_numeric
     0 2021-06-29  morning               0
     1 2021-06-29  daytime               1
     2 2021-06-29  evening               2
     3 2021-07-23  morning               0
     4 2021-07-23  daytime               1
```

This thorough segment-based expansion of the dataset set the stage for a detailed mood pattern analysis. The data now reflected a clearer picture of how different times of the day could potentially influence mood swings.

```
[ ]: daily_summary_chopped.to_csv('/Users/mirayozcan/Desktop/daily_summary_chopped.
     ↪csv', index=False)
```

1.3.13 Pre-Processing Step 3: Labeling the Updated Daily Segment-Based Dataset

A significant modification in this phase of pre-processing was the transition from a multi-class to a binary classification system. The initial four mood categories were reorganized into two principal groups to amplify the sample size for each category, thereby enhancing the robustness of the subsequent machine learning models.

The categories were consolidated as follows: 1. **Negative:** A combination of the previous 'Overwhelmed' and 'Busy' labels. This category encompasses days marked by excessive demands or packed schedules, potentially leading to stress or negative feelings.

2. **Positive:** Uniting the former 'Relaxed' and 'Productive' labels. Days classified under this umbrella typically balance leisure and productivity, contributing to a positive outlook.

The dataset, now segmented by day parts, required relabeling to reflect this binary scheme. The reclassification aimed to align with the improved data strategy suggested by feedback, ensuring each mood state was supported by a sufficient number of samples.

```
[ ]: # Re-importing the labeled dataset

daily_summary_chopped_labeled = pd.read_csv('/Users/mirayozcan/Desktop/
 ↪daily_summary_chopped_labeled.csv')
daily_summary_chopped_labeled.head()
```

```
[ ]:        date  segment event_ids  total_duration  avg_duration  n_events  \
0  6/29/2021  morning         0               0           0.0         0
1  6/29/2021  daytime         0               0           0.0         0
2  6/29/2021  evening       325              15          15.0         1
3  7/23/2021  morning         0               0           0.0         0
4  7/23/2021  daytime         0               0           0.0         0

   total_attendees  avg_attendees  start_first_event  end_last_event  \
0                0            0.0                 -1              -1
1                0            0.0                 -1              -1
2                2            2.0                 20              20
3                0            0.0                 -1              -1
4                0            0.0                 -1              -1

   day_of_month  day_of_week  has_event  segment_numeric      mood
0            29            2          0                0  Positive
1            29            2          0                1  Positive
2            29            2          1                2  Positive
3            23            5          0                0  Positive
4            23            5          0                1  Positive
```

By simplifying the mood categories, the dataset not only becomes more manageable but also allows for more straightforward interpretations of the results from predictive modeling. This binary classification lays the groundwork for a focus on discerning between broadly positive and negative days, which could be particularly useful for applications aimed at general mood improvement strategies.

1.3.14 Pre-Processing Step 4: Incorporating Textual Data from the Event Descriptions

In the quest to harness the full potential of the dataset, the decision was made to enrich the dataset with descriptive textual data derived from event summaries. This step was crucial as textual data often hold intricate nuances that can significantly influence classification outcomes.

1. **Filling Missing Descriptions:** To mitigate the lack of descriptions, a generative language model (ChatGPT-4) was employed. For events missing descriptions, new ones were generated, leveraging the succinct summaries as a seed to ensure relevance and context.

2. **Paraphrasing for Uniformity:** Consistency in the language used across descriptions was attained through paraphrasing. This harmonization aimed to reduce non-informative variance, thus providing the model with a coherent language style and potentially equal sequence lengths.

3. **Ensuring Descriptive Coverage:** The end goal was to guarantee that every event in the dataset possessed a description, thereby maximizing the textual information available for model training.

A batch of event titles was fed into ChatGPT-4 with a prompt tailored to generate informative and concise descriptions, approximately 15 words in length. This approach not only addressed the absence of descriptions but also revitalized existing ones for uniformity and richness.

Sample GPT-4 Prompt for Textual Data Generation: Given the existing summaries (titles) of my personal calendar data for the 10 data points pasted below, can you generate informative descriptions of the events by utilizing their summaries? The updated descriptions should be approximately 15 words long and appropriate for training purposes for machine learning models, specifically for text-data-based classification tasks. When you have the updated descriptions, please prepare them in a format that enables me to copy your response and paste them into the "description" column of the associated data points. Although the summary might be the same for each event, strive for a unique description to increase variability in the training data. However, when this is not possible, two events with the exact same summaries could also have the same description.

```
[ ]: # Importing the enhanced dataset with new event descriptions
     df_chopped_enhanced = pd.read_csv('/Users/mirayozcan/Desktop/
      ↪events_chopped_enhanced.csv')

     # Display the first few rows to verify the updated descriptions
     df_chopped_enhanced.head()
```

```
[ ]:    event_id       date start_time  end_time  duration  n_attendees  \
     0       1.0  8/25/2021    7:15:00   7:30:00      15.0          2.0
     1       2.0  9/27/2021   23:00:00   0:00:00      60.0          1.0
     2       3.0  10/4/2021   16:00:00  19:30:00     210.0          1.0
     3       4.0  10/5/2021    3:30:00   4:30:00      60.0          1.0
     4       5.0  10/5/2021   14:45:00  15:00:00      15.0          1.0

                                              description              summary  \
     0  Join İsil and Miray Özcan for an interactive w…  İsil and Miray Özcan
     1  Participate in a comprehensive Time Management…   Time Management Lab
     2  Engage in both Social Science 50 and Computer …  SS50 AND CS50 CLASSES
     3  Indulge in a relaxation session to unwind and …        Relaxation Time
     4  Start your day energetically with a morning wa…                Wake Up

       is_within_segment  segment  start_hour  end_hour
```

```
0              True   morning         7.0        7.0
1              True   evening        23.0        0.0
2              True   evening        16.0       19.0
3              True   morning         3.0        4.0
4              True   daytime        14.0       15.0
```

This process was not only about filling gaps in the dataset but also about enriching the data's quality to serve as a solid foundation for the models to learn more effectively from the textual nuances. It represented a blend of automated AI capabilities and strategic data preparation, tailored for advanced machine learning applications.

1.3.15 Pre-Processing Step 5: Enhancing the Daily Segment-Based Dataset with Event Descriptions

The aim of this preprocessing step is to amalgamate the descriptive textual content into the day segment-based aggregation. By doing so, we elevate the dataset's ability to convey not just the quantitative but also the qualitative nuances of daily schedules.

1. **Empty Descriptions:** For segments without events (has_event = 0), the absence of activity is explicitly stated with "No event descriptions." This clear demarcation ensures that the model recognizes segments of inactivity.
2. **Populated Descriptions:** For segments with events (has_event = 1), descriptions from events_chopped_enhanced are collated to represent the day's narrative. This methodical gathering of text paints a richer picture of the segment's characteristics.

[ ]:
```python
# Loading the datasets

events_chopped_enhanced = pd.read_csv('/Users/mirayozcan/Desktop/
 ↪events_chopped_enhanced.csv')
daily_summary_chopped_labeled = pd.read_csv('/Users/mirayozcan/Desktop/
 ↪daily_summary_chopped_labeled.csv')

# Creating a dictionary to map event IDs to their descriptions
event_description_mapping = dict(zip(events_chopped_enhanced.event_id,␣
 ↪events_chopped_enhanced.description))

# Defining a function to concatenate event descriptions for each day segment
def get_event_descriptions(row, event_description_mapping):
    if row['has_event'] == 1:
        event_ids = [int(e) for e in str(row['event_ids']).split(',')]
        descriptions = [event_description_mapping.get(event_id, 'No description␣
 ↪available.') for event_id in event_ids]
        return ' '.join(descriptions)
    else:
        return 'No event descriptions.'

# Applying the function to compile descriptions for the daily segments
```

```
daily_summary_chopped_labeled['events_descriptions'] =␣
↪daily_summary_chopped_labeled.apply(
    lambda row: get_event_descriptions(row, event_description_mapping), axis=1
)

# Verifying the process by displaying the head of the dataframe
daily_summary_chopped_labeled[['date', 'segment', 'events_descriptions']].head()
```

[ ]:
```
        date  segment                                events_descriptions
0  6/29/2021  morning                             No event descriptions.
1  6/29/2021  daytime                             No event descriptions.
2  6/29/2021  evening  Engage in activities with Cecile S for collabo…
3  7/23/2021  morning                             No event descriptions.
4  7/23/2021  daytime                             No event descriptions.
```

This approach of textual data enhancement is not only about appending strings but introducing context and substance that can dramatically influence model training. Textual descriptions offer an insight that could be pivotal in discerning the nature of the day and thus, the mood associated with it.

The output is a finely curated dataset where each day segment is characterized by a descriptive account of the activities or the notable lack thereof. This dataset is now prepped for models that can integrate and interpret both numerical and textual data to predict outcomes with greater depth and accuracy.

[ ]:
```
# Full visualization

daily_summary_chopped_labeled.head()
```

[ ]:
```
        date  segment event_ids  total_duration  avg_duration  n_events  \
0  6/29/2021  morning         0               0           0.0         0
1  6/29/2021  daytime         0               0           0.0         0
2  6/29/2021  evening       325              15          15.0         1
3  7/23/2021  morning         0               0           0.0         0
4  7/23/2021  daytime         0               0           0.0         0

   total_attendees  avg_attendees  start_first_event  end_last_event  \
0                0            0.0                 -1              -1
1                0            0.0                 -1              -1
2                2            2.0                 20              20
3                0            0.0                 -1              -1
4                0            0.0                 -1              -1

   day_of_month  day_of_week  has_event  segment_numeric      mood  \
0            29            2          0                0  Positive
1            29            2          0                1  Positive
2            29            2          1                2  Positive
3            23            5          0                0  Positive
```

```
4          23          5          0               1  Positive

                              events_descriptions
0                           No event descriptions.
1                           No event descriptions.
2  Engage in activities with Cecile S for collabo…
3                           No event descriptions.
4                           No event descriptions.
```

As a recap, here is a glossary for column name descriptions in the final version of our dataset:

- `date`: The date when the events took place. Data type: `Date`.
- `segment`: A part of the day categorized as morning, daytime, or evening. Data type: `String`.
- `event_ids`: Comma-separated IDs of events that occurred within a segment. Data type: `String`.
- `total_duration`: The sum of the durations (in minutes) of all events in a segment. Data type: `Integer`.
- `avg_duration`: The average duration (in minutes) of events in a segment. Data type: `Float`.
- `n_events`: The total number of events within a segment. Data type: `Integer`.
- `total_attendees`: The total number of attendees for all events in a segment. Data type: `Integer`.
- `avg_attendees`: The average number of attendees per event in a segment. Data type: `Float`.
- `start_first_event`: The starting hour of the first event in a segment (if any), `-1` indicates no event. Data type: `Integer`.
- `end_last_event`: The ending hour of the last event in a segment (if any), `-1` indicates no event. Data type: `Integer`.
- `day_of_month`: The day of the month extracted from the `date`. Data type: `Integer`.
- `day_of_week`: The day of the week extracted from the `date`, where `1` is Monday and `7` is Sunday. Data type: `Integer`.
- `has_event`: Indicator of whether at least one event occurred in a segment (`1` for yes, `0` for no). Data type: `Integer`.
- `segment_numeric`: Numerical encoding of the day segment (`0` for morning, `1` for daytime, `2` for evening). Data type: `Integer`.
- `mood`: The mood label for the segment, categorized as Positive or Negative. Data type: `String`.
- `events_descriptions`: Descriptions of events during the segment, or "No event descriptions." when empty. Data type: `String`.

```
[ ]: # Saving the updated dataset with event descriptions to a CSV file
daily_summary_chopped_labeled.to_csv('/Users/mirayozcan/Desktop/
 ↪daily_summary_chopped_labeled_enhanced.csv', index=False)
```

1.3.16 Further Cleaning and Pre-Processing on Textual Data (event_descriptions Column) for Model Preparation

The textual data within the `event_descriptions` column of our dataset holds a wealth of information that can significantly influence the performance of machine learning models. However, this raw textual data often contains noise – such as special characters, numbers, and common words – that can obscure the meaningful patterns models might learn from. Therefore, further cleaning and preprocessing of this textual data is crucial to prepare it effectively for model training.

To enhance the quality of the textual data and ensure consistency across the dataset, the following preprocessing steps were employed:

- **Placeholder Removal**: It was identified that “No event descriptions.” serves as a placeholder text for events without descriptions. Since this does not provide meaningful information for modeling, it’s treated as noise and removed.
- **Case Normalization**: Text is converted to lowercase to ensure that the same words in different cases are treated as identical tokens. This step helps in reducing the complexity of the data.
- **Punctuation and Special Characters Removal**: Punctuation, special characters, and numbers were removed as these elements are typically not useful for text representation learning and are considered noise.
- **Tokenization**: The process of breaking text into individual words or tokens is essential since machine learning models operate on numerical data. Tokenization serves as the first step in transforming text into a format that models can understand.
- **Stopwords Removal**: Common words like “and”, “is”, “in”, etc., which usually don’t carry significant meaning, were removed. This helps in reducing the feature space size, allowing models to focus on more informative aspects of the text.
- **Stemming/Lemmatization**: This step involves reducing words to their root form. For instance, “running” and “runs” would both be reduced to “run”. It helps in consolidating different forms of the same word, thus simplifying the feature space and aiding the model’s generalization capability.

[ ]:
```
# Libraries

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
import pandas as pd
import re
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from nltk.stem import PorterStemmer
```

[ ]:
```
# Dataset

df = pd.read_csv('/Users/mirayozcan/Desktop/
 ↪daily_summary_chopped_labeled_enhanced.csv')

# Function to preprocess text data

def preprocess_text(text):
    if text == "No event descriptions.":
        return ""
    text = text.lower()
    text = re.sub(r'[^a-zA-Z\s]', '', text)
    tokens = word_tokenize(text)
    stop_words = set(stopwords.words('english'))
    filtered_tokens = [word for word in tokens if not word in stop_words]
    stemmer = PorterStemmer()
    stemmed_tokens = [stemmer.stem(word) for word in filtered_tokens]
    return " ".join(stemmed_tokens)

# Applying preprocessing
df['events_descriptions_processed'] = df['events_descriptions'].
  ↪apply(preprocess_text)

# Displaying preprocessed text
df[['events_descriptions', 'events_descriptions_processed']]
```

[ ]:

```
                                     events_descriptions  \
0                                 No event descriptions.
1                                 No event descriptions.
2     Engage in activities with Cecile S for collabo…
3                                 No event descriptions.
4                                 No event descriptions.
…                                                  …
1519                              No event descriptions.
1520  Participate in introspective Q&A session on jo…
1521                              No event descriptions.
1522                              No event descriptions.
1523  Participate in introspective Q&A session on jo…

                          events_descriptions_processed
0
1
2                       engag activ cecil collabor effort
3
4
…                                                  …
1519
1520  particip introspect qa session job readi strat…
1521
1522
1523  particip introspect qa session job readi strat…

[1524 rows x 2 columns]
```

[ ]:

```
df.to_csv('/Users/mirayozcan/Desktop/daily_summary_chopped_labeled_enhanced.
  ↪csv', index=False)
```

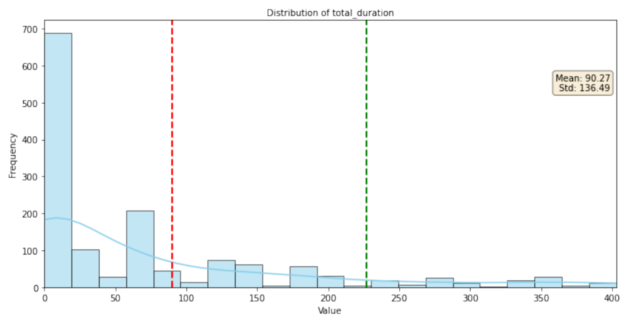
1.3.17 Exploratory Analysis on the Second Pipeline Data

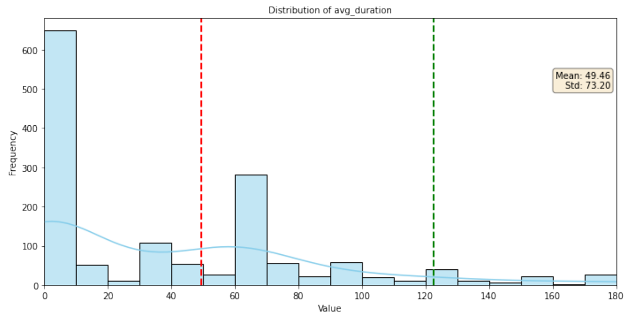
1- Numerical Data Exploration

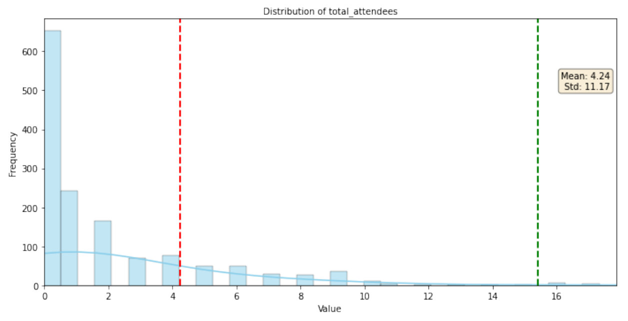
Histograms and Descriptive Statistics

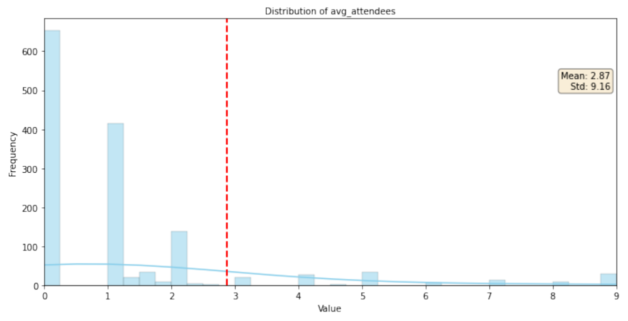
```
[ ]: # Updated list of continuous variable names

continuous_vars = [
    'total_duration', 'avg_duration',
    'total_attendees', 'avg_attendees', 'start_first_event',
    'end_last_event']

# Plot updated histograms for each continuous variable
fig, axes = plt.subplots(len(continuous_vars), 1, figsize=(10, 5 *␣
 ↪len(continuous_vars)))

for i, var in enumerate(continuous_vars):
    # Determine the 95th percentile for the current variable
    percentile_95 = np.percentile(daily_summary_chopped_labeled[var].dropna(),␣
 ↪95)

    sns.histplot(data=daily_summary_chopped_labeled, x=var, kde=True,␣
 ↪ax=axes[i], color='skyblue', edgecolor='black', binrange=(0, percentile_95))
    axes[i].set_title(f'Distribution of {var}', fontsize=10)
    axes[i].set_ylabel('Frequency')
    axes[i].set_xlabel('Value')
    axes[i].set_xlim(left=0, right=percentile_95)  # Set the x-axis limit to␣
 ↪the 95th percentile

    # Calculate mean and standard deviation
    mean_val = daily_summary_chopped_labeled[var].mean()
    std_val = daily_summary_chopped_labeled[var].std()

    # Plot the mean line
    axes[i].axvline(mean_val, color='red', linestyle='dashed', linewidth=2)
    # Plot the standard deviation lines (mean +/- std)
    axes[i].axvline(mean_val + std_val, color='green', linestyle='dashed',␣
 ↪linewidth=2)
    axes[i].axvline(mean_val - std_val, color='green', linestyle='dashed',␣
 ↪linewidth=2)

    # Print mean and standard deviation
    axes[i].text(0.99, 0.8, f'Mean: {mean_val:.2f}\nStd: {std_val:.2f}',␣
 ↪transform=axes[i].transAxes,
            verticalalignment='top', horizontalalignment='right', fontsize=10,
            bbox=dict(boxstyle='round', facecolor='wheat', alpha=0.5))
```

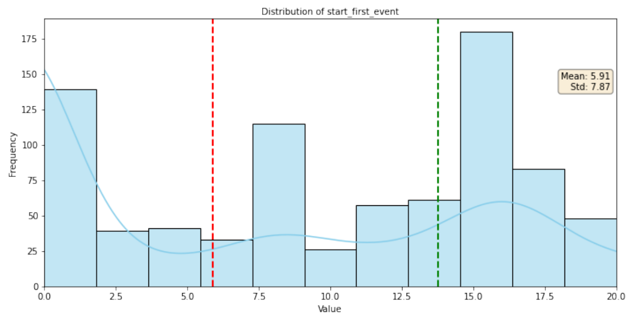
```
plt.tight_layout()
plt.show()
```

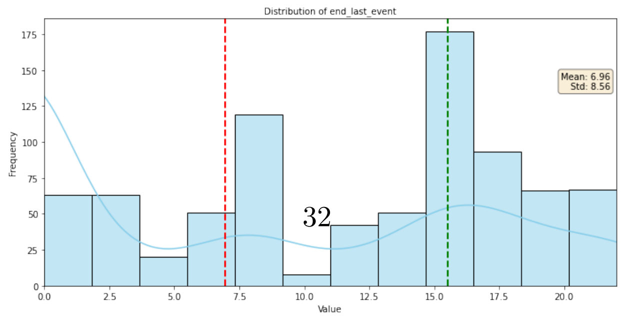

Distribution of total_duration
Frequency
Value
Mean: 90.27
Std: 136.49


Distribution of avg_duration
Frequency
Value
Mean: 49.46
Std: 73.20


Distribution of total_attendees
Frequency
Value
Mean: 4.24
Std: 11.17


Distribution of avg_attendees
Frequency
Value
Mean: 2.87
Std: 9.16

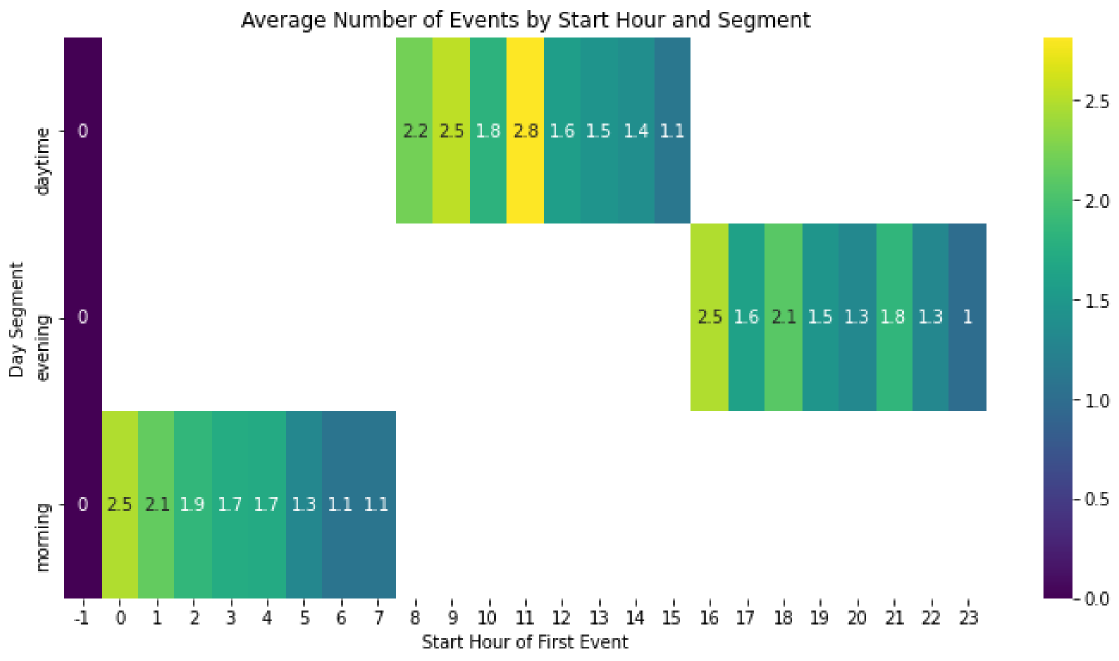

Distribution of start_first_event
Frequency
Value
Mean: 5.91
Std: 7.87


Distribution of end_last_event
Frequency
Value
Mean: 6.96
Std: 8.56

1. **Total Duration of Events**: This histogram shows the distribution of the total duration of events within each day segment. The mean duration is around 90 minutes, but the significant standard deviation suggests a broad range of activity levels, from quick tasks to extended engagements.
2. **Average Duration of Events**: The average duration per event is visualized here, indicating that most events tend to be brief. A mean around 49 minutes and a wide spread reflect diverse event lengths, from short meetings to lengthy workshops or social gatherings.
3. **Total Number of Attendees**: This histogram depicts the total number of attendees per segment. A lower mean suggests that segments often include smaller, perhaps more focused or personal events. The spread points to occasional larger gatherings that significantly deviate from the norm.
4. **Average Number of Attendees**: Focusing on the average number of attendees per event, this histogram illustrates a tendency towards intimate settings, with most events hosting fewer individuals. The standard deviation, however, hints at the sporadic presence of more populous events.
5. **Start Time of First Event**: The distribution of when the first event of the day starts reveals a concentration in the earlier hours. The average start time indicates that days often begin by mid-morning, yet there's enough variation to account for early risers or late starters.
6. **End Time of Last Event**: Examining the end time of the last event, this histogram shows most days conclude by early evening. The mean end time reflects a typical end-of-workday pattern, though the spread includes days that stretch into the night, indicating flexible end times.

Heatmap of Average Event Counts Across Day Segments and Start Times

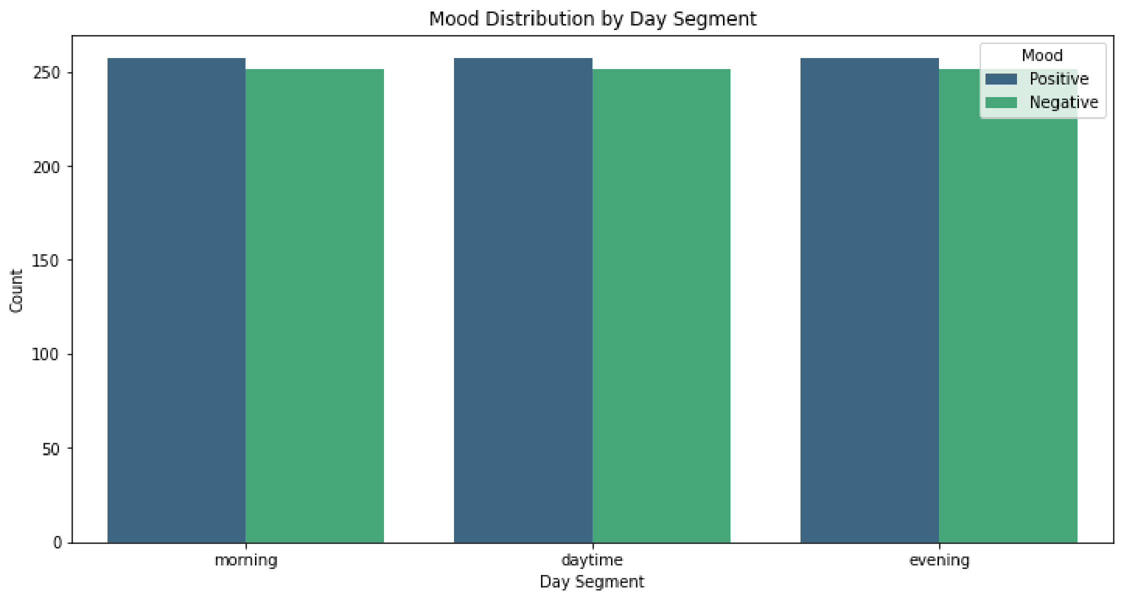
[ ]:
```python
# Create a pivot table with hours and segments as dimensions
hourly_pivot = daily_summary_chopped_labeled.pivot_table(
    index='segment',
    columns='start_first_event',
    values='n_events',
    aggfunc='mean'
)

# Create a heatmap of the average number of events
plt.figure(figsize=(12, 6))
sns.heatmap(hourly_pivot, annot=True, cmap='viridis')
plt.title('Average Number of Events by Start Hour and Segment')
plt.xlabel('Start Hour of First Event')
plt.ylabel('Day Segment')
plt.show()
```



The heatmap provides a visualization of the average number of events that start at various hours throughout the day, broken down by day segment (morning, daytime, evening). Each cell's color intensity reflects the average event count for that time segment, with warmer colors representing higher averages. In this representation, it appears that the daytime segment tends to have a higher average number of events starting around mid-morning to early afternoon. Contrastingly, the morning and evening segments show a wider distribution of start times, indicating more variability in when events begin. Notably, there is an absence of events starting at certain hours during the day, which is represented by the darker cells (e.g., no events that start at hour 0 for daytime and evening segments).

Comparative Mood Distribution Across Different Day Segments

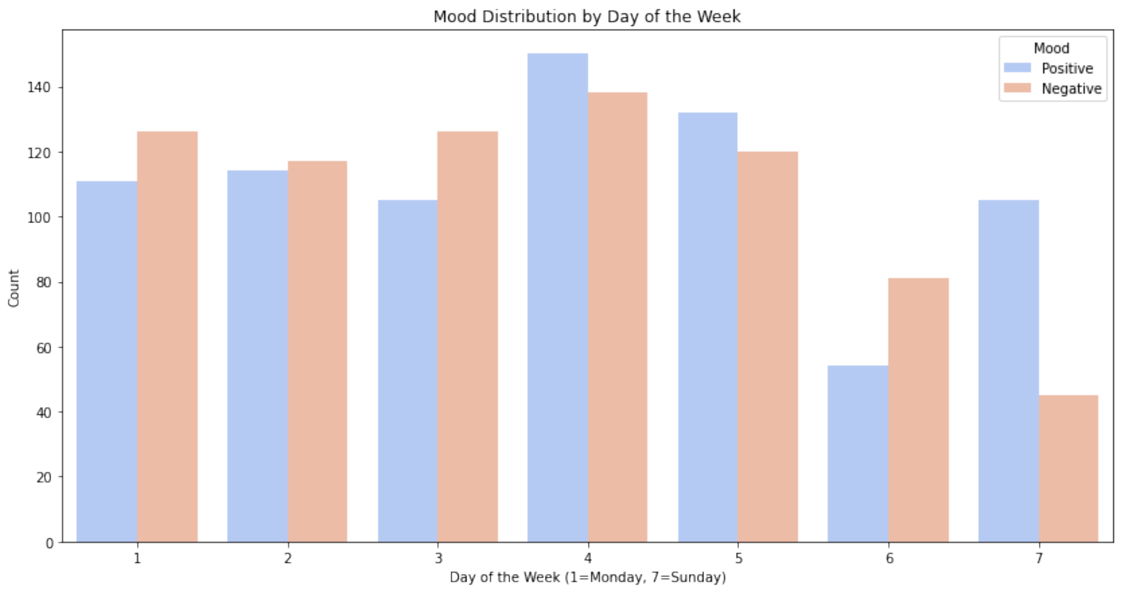
```
[ ]: plt.figure(figsize=(12, 6))
     sns.countplot(data=daily_summary_chopped_labeled, x='segment', hue='mood',␣
      ↪palette='viridis')
     plt.title('Mood Distribution by Day Segment')
     plt.xlabel('Day Segment')
     plt.ylabel('Count')
     plt.legend(title='Mood')
     plt.show()
```



This bar chart illustrates the distribution of mood states—positive and negative—across three different segments of the day: morning, daytime, and evening. Each segment's bar is bifurcated into two colors representing the count of positive (dark blue) and negative (green) moods recorded. The nearly equal height of bars across all segments suggests a uniform distribution of mood states throughout the day. Interestingly, there is a slightly higher occurrence of positive moods during the evening segment, which could suggest that people tend to end their day on a more positive note or feel a sense of relief as the day concludes. In contrast, the morning and daytime segments show a very close count of positive and negative moods, indicating a balanced emotional state during earlier parts of the day.

Daily Mood Trends Across the Week

```
[ ]: plt.figure(figsize=(14, 7))
     sns.countplot(data=daily_summary_chopped_labeled, x='day_of_week', hue='mood',␣
       ↪palette='coolwarm', dodge=True)
     plt.title('Mood Distribution by Day of the Week')
     plt.xlabel('Day of the Week (1=Monday, 7=Sunday)')
     plt.ylabel('Count')
     plt.legend(title='Mood', loc='upper right')
     plt.show()
```

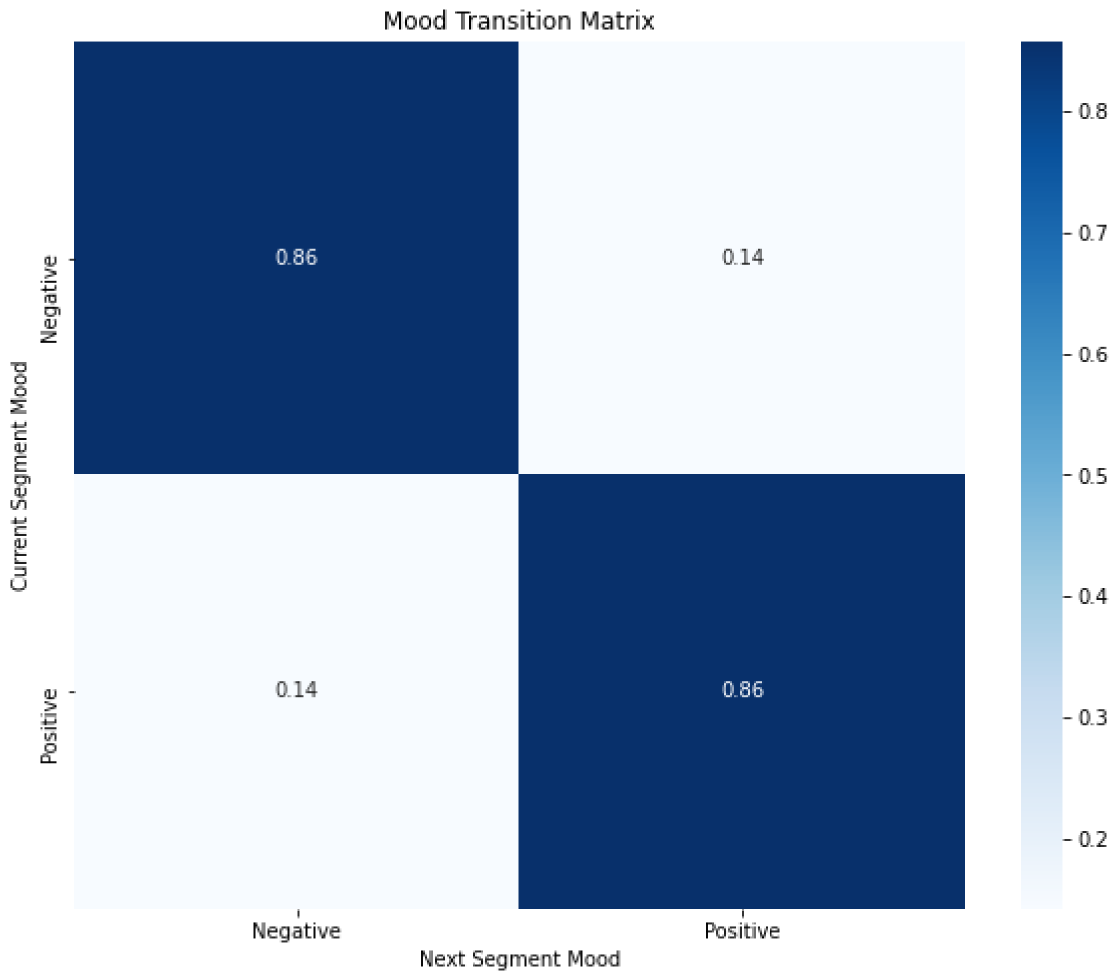


This chart presents the mood distribution across day segments throughout the week. Each day is marked from 1 (Monday) to 7 (Sunday), displaying the count of positive (blue) and negative (salmon) moods recorded. The mid-week days show a higher prevalence of positive moods, possibly reflecting midweek productivity or routine stability. In contrast, the weekends, particularly Sunday, exhibit a surge in negative moods, which could suggest end-of-weekend blues or anticipation of the upcoming week.

Segment-to-Segment Mood Stability Heatmap

```
[ ]: # Create a new DataFrame that shifts the mood column
     # Assuming each row in daily_summary_chopped_labeled is a day segment ordered␣
      ↪chronologically
     daily_summary_chopped_labeled['next_mood'] = daily_summary_chopped_labeled.
      ↪groupby(['date'])['mood'].shift(-1)

     # For evening segments, next_mood should be the mood of the following morning
     # This assumes that the data is sorted by date and segment
     is_evening = daily_summary_chopped_labeled['segment'] == 'evening'
     following_morning = daily_summary_chopped_labeled['segment'].shift(-1) ==␣
      ↪'morning'
     next_day = daily_summary_chopped_labeled['date'] !=␣
      ↪daily_summary_chopped_labeled['date'].shift(-1)

     # Where it's an evening segment and the following row is a morning of a next␣
      ↪day,
     # shift the mood from the following row
     daily_summary_chopped_labeled.loc[is_evening & following_morning & next_day,␣
      ↪'next_mood'] = daily_summary_chopped_labeled['mood'].shift(-1)
```

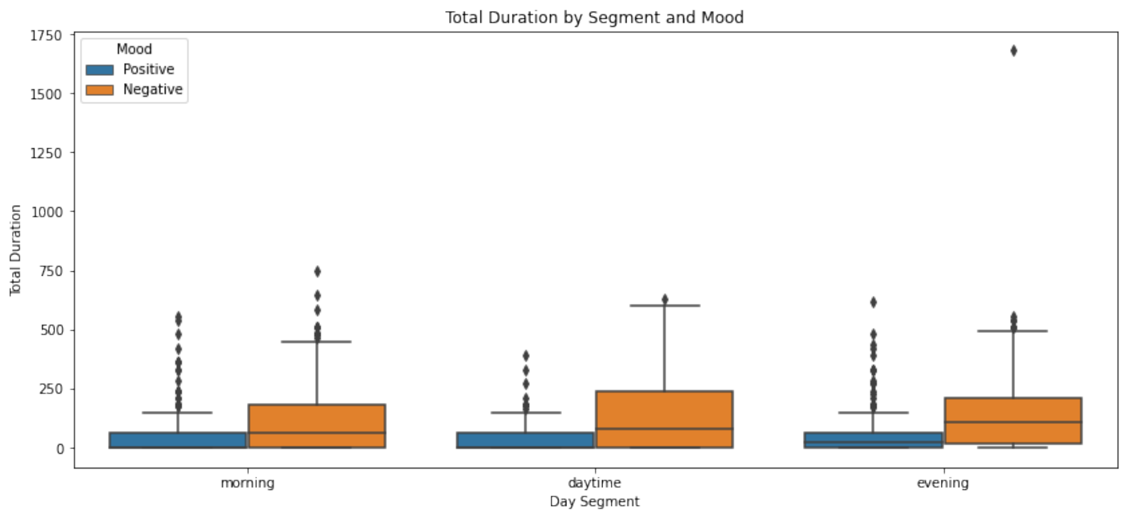
```
# Drop the rows where next_mood is NaN, which will be the last segment of the␣
↪dataset
daily_summary_chopped_labeled.dropna(subset=['next_mood'], inplace=True)

# Calculate the mood transition matrix
mood_transition_matrix = pd.crosstab(daily_summary_chopped_labeled['mood'],
                                     daily_summary_chopped_labeled['next_mood'],
                                     normalize='index')

# Plotting the mood transition matrix
plt.figure(figsize=(10, 8))
sns.heatmap(mood_transition_matrix, annot=True, cmap="Blues", fmt=".2f")
plt.title('Mood Transition Matrix')
plt.xlabel('Next Segment Mood')
plt.ylabel('Current Segment Mood')
plt.show()
```



The heatmap visualizes the stability of mood from one segment to the next within a day. High values along the diagonal indicate a tendency for moods to persist; with both negative and positive states showing about 86% continuity. This suggests a strong mood inertia, where the current emotional state is likely to remain the same in the subsequent segment. Transitioning from a negative to a positive mood or vice versa is less common, reflecting the resilience of mood states against rapid shifts within the day's structure.

Total Duration of Activities and Mood Correlation by Day Segment

```
[ ]: plt.figure(figsize=(14, 6))
     sns.boxplot(data=daily_summary_chopped_labeled, x='segment',␣
      ↪y='total_duration', hue='mood')
     plt.title('Total Duration by Segment and Mood')
     plt.xlabel('Day Segment')
     plt.ylabel('Total Duration')
     plt.legend(title='Mood')
     plt.show()
```

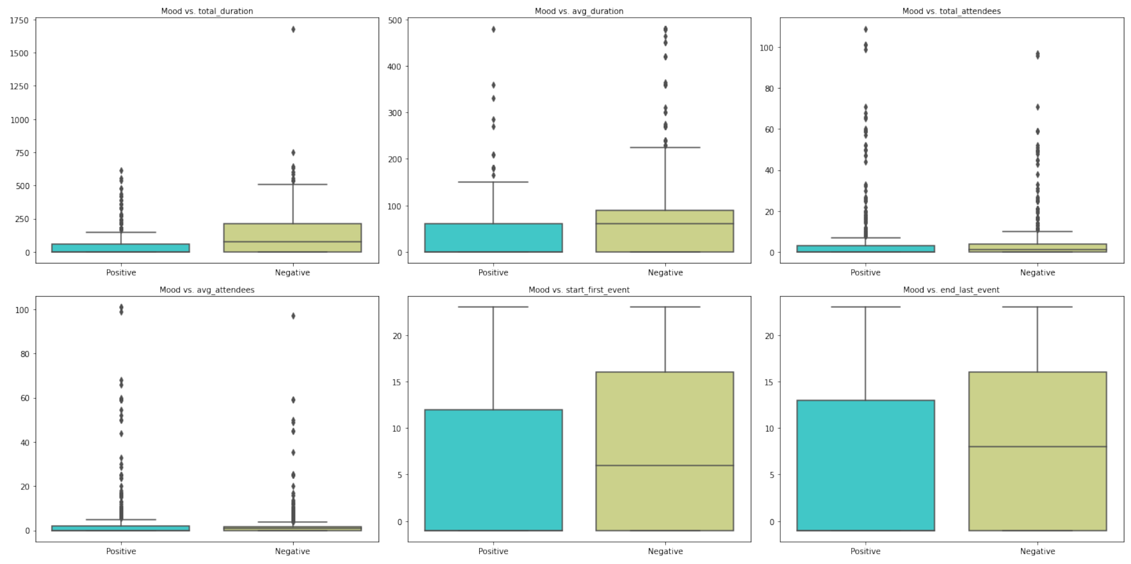


This boxplot analysis indicates that the total duration of activities doesn't significantly differentiate moods within morning and evening segments, suggesting these periods have a balanced mixture of positive and negative moods regardless of activity length. However, during the daytime, there's a noticeable variation in activity duration between moods, with negative moods associated with longer activity durations. This might imply that busier midday schedules could contribute to a negative mood, or conversely, that a negative mood perceives or results in the day feeling more loaded.

Comparative Mood Analysis Across Event Features

```
[ ]: # Adjusting the subplot grid to accommodate the number of continuous columns
     n_rows = (len(continuous_vars) + 2) // 3
```

```
fig, axes = plt.subplots(n_rows, 3, figsize=(20, 5*n_rows))  # Adjust figsize␣
↪accordingly
axes = axes.flatten()

# Loop through each continuous column to create boxplots
for ax, column in zip(axes, continuous_vars):
    sns.boxplot(x='mood', y=column, data=daily_summary_chopped_labeled, ax=ax,␣
↪palette='rainbow')
    ax.set_title(f'Mood vs. {column}', fontsize=10)
    ax.set_xlabel('')
    ax.set_ylabel('')

plt.tight_layout()
plt.show()
```

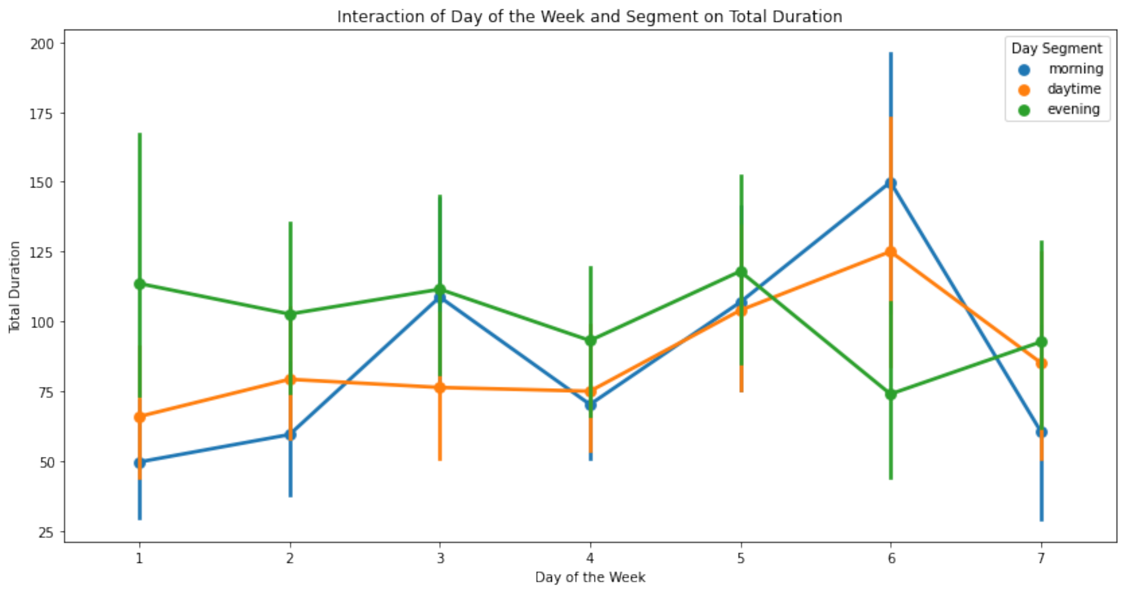


This set of boxplots provides an insightful comparison across various event features segmented by mood. The 'total_duration' feature shows a wider range for negative moods, suggesting a possible link between longer event durations and more negative mood assessments. For 'avg_duration' and 'total_attendees', both moods display similar medians, indicating that these factors may not be as influential on mood. However, the starting and ending times of events ('start_first_event' and 'end_last_event') present noticeable differences; negative moods are linked with earlier starts and later ends, hinting at a potential preference for more free time during mornings and evenings for a positive mood.

Weekly Dynamics: Segment Impact on Total Duration

```
[ ]: plt.figure(figsize=(14, 7))
     sns.pointplot(data=daily_summary_chopped_labeled, x='day_of_week',␣
     ↪y='total_duration', hue='segment')
```

```
plt.title('Interaction of Day of the Week and Segment on Total Duration')
plt.xlabel('Day of the Week')
plt.ylabel('Total Duration')
plt.legend(title='Day Segment')
plt.show()
```

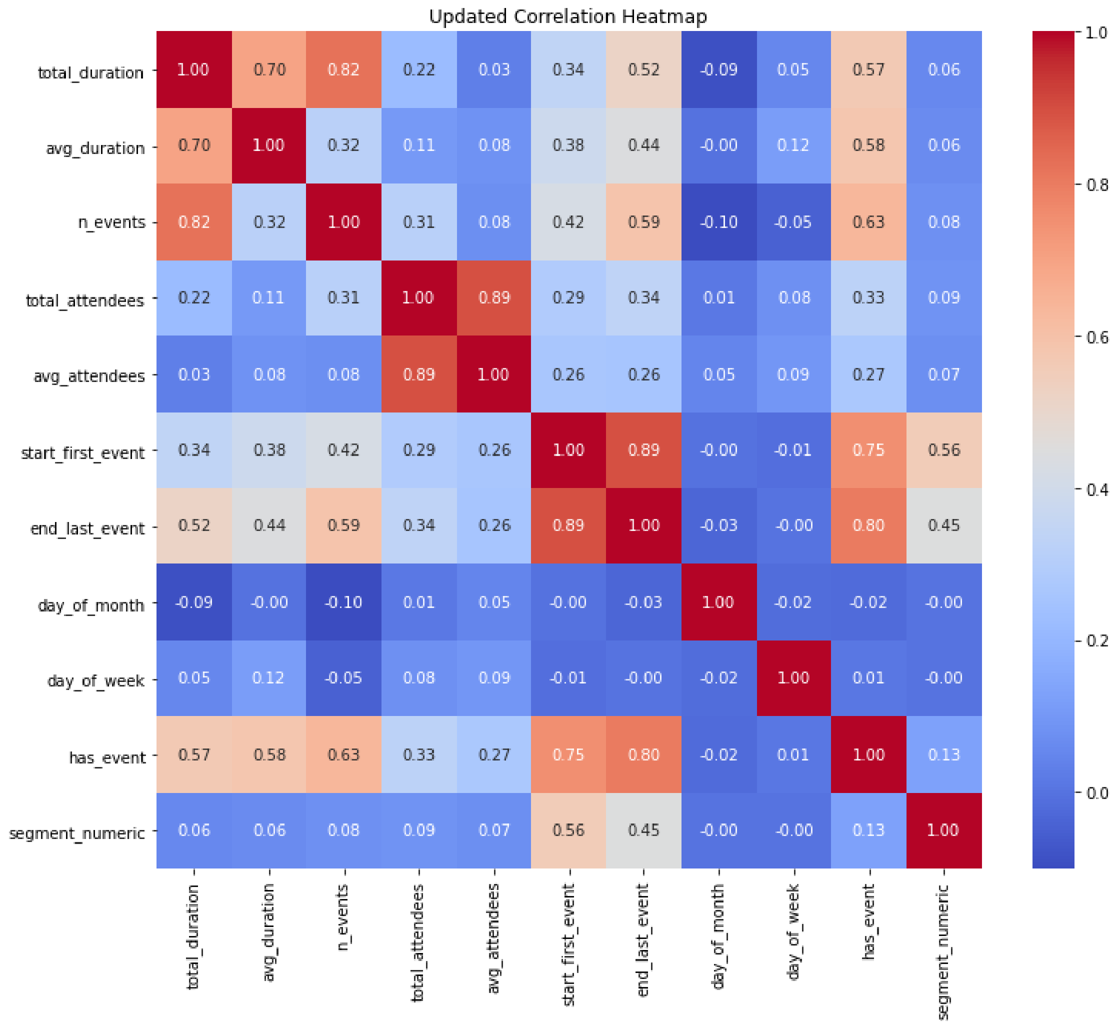


This point plot reveals the dynamic interplay between day segments and total event duration throughout the week. Morning durations remain relatively consistent, suggesting a stable start to the day, while daytime and evening durations fluctuate more notably. Midweek spikes in daytime duration may reflect peak productivity or scheduled commitments, with a sharp increase in evening durations towards the weekend, potentially indicating social or leisure activities. Interestingly, the end-of-week drop in daytime durations could signify a collective winding down, transitioning into quieter evenings.

Correlation and Pairwise Relationships Amongst Event Characteristics

[ ]:
```
# Including the segment_numeric in the correlation analysis

numeric_vars = [
    'total_duration', 'avg_duration', 'n_events',
    'total_attendees', 'avg_attendees', 'start_first_event',
    'end_last_event', 'day_of_month', 'day_of_week', 'has_event',
    'segment_numeric'
]

corr_vars = daily_summary_chopped_labeled[numeric_vars]
corr_matrix = corr_vars.corr()
```

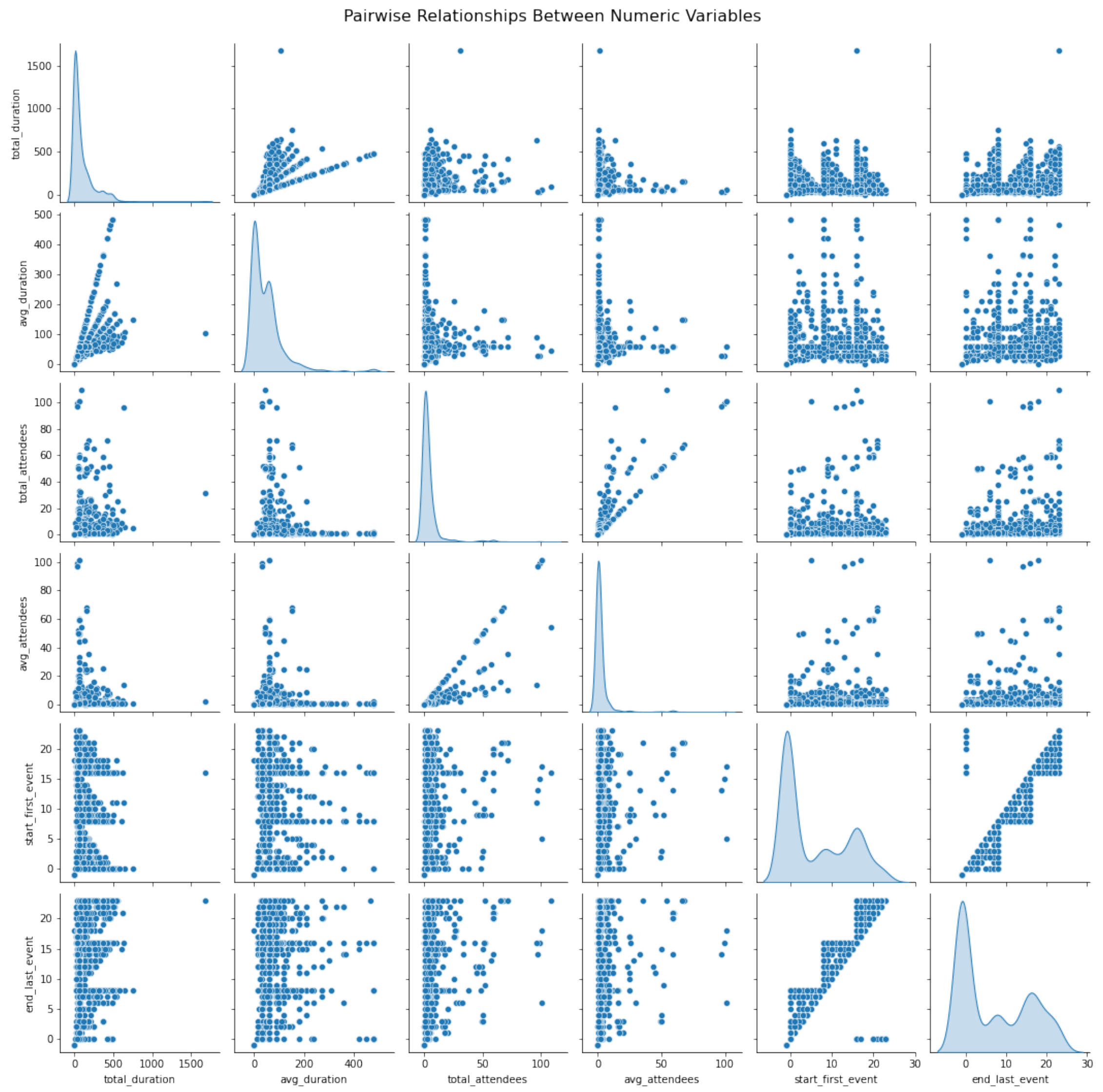
```
plt.figure(figsize=(12, 10))
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', fmt='.2f')
plt.title('Updated Correlation Heatmap')
plt.show()
```



The heatmap visualizes the strength of correlation between various daily activity metrics. Notable is the strong positive relationship between total duration, average duration, and number of events, indicating that more events generally lead to longer days. A similar strong positive correlation between the start and end times of events suggests a consistent daily routine. Attendee metrics display a weaker correlation with other variables, implying that the number of people at events does not significantly dictate the day's structure. Interestingly, the day of the month and the week show negligible influence on daily activities, emphasizing a consistent daily rhythm regardless of the time within the month or week. The presence of events ('has_event') is moderately linked with higher event counts and durations, highlighting that active segments are distinctly busier.

Pairplot of Numeric Continuous Variables

```
[ ]: # Generate the pairplot
     sns.pairplot(daily_summary_chopped_labeled[continuous_vars], diag_kind='kde')

     # Adjust the height of each subplot
     plt.subplots_adjust(top=0.95)
     plt.suptitle('Pairwise Relationships Between Numeric Variables', fontsize=16)
     plt.show()
```

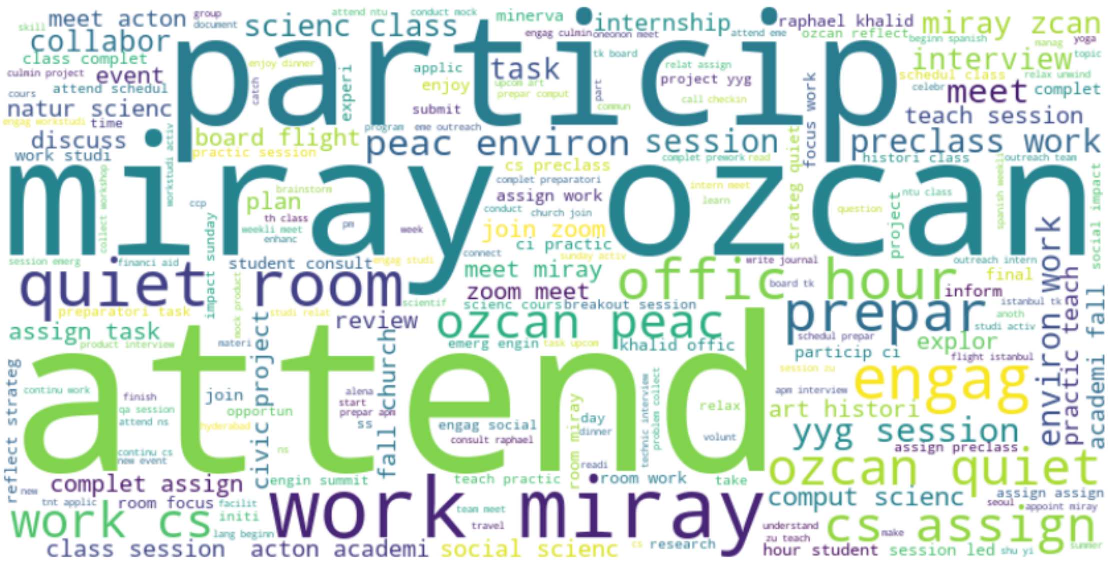


The scatterplot matrix presents a web of relationships between daily event variables. Diagonal plots show distribution densities, highlighting that total durations, event counts, and attendee numbers typically stay low, with fewer instances of higher values. Off-diagonal plots reveal clear positive correlations, especially between the number of events and total duration, signifying that more events usually extend the day's schedule. Start and end times of events are strongly linear, suggesting a consistent event timeline from day to day. The scatter among attendee-related variables suggests variability in meeting sizes, independent of the day's structure. Sparse points in upper regions indicate outlier days with unusual activity, either in duration or attendee count.

2- Textual Data Exploration

Word Cloud of Event Descriptions

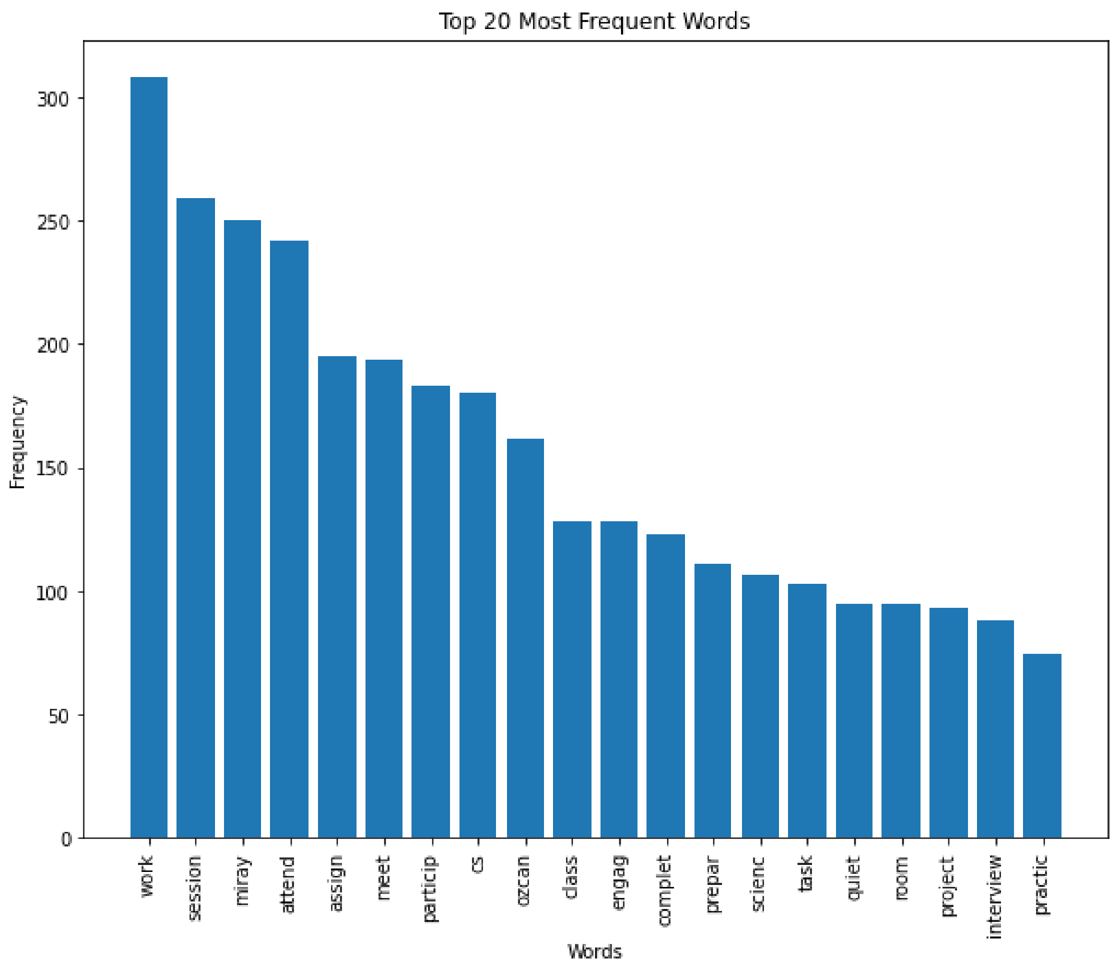
```
[ ]: from wordcloud import WordCloud

     # Combine all preprocessed descriptions into one large string
     corpus = ' '.join(df['events_descriptions_processed'])

     # Generate word cloud
     wordcloud = WordCloud(width=800, height=400, background_color='white').
      ↪generate(corpus)

     # Display the word cloud
     plt.figure(figsize=(15, 10))
     plt.imshow(wordcloud, interpolation='bilinear')
     plt.axis('off')
     plt.show()
```



This word cloud presents a visual summary of the most prevalent terms within the event descriptions, with larger font sizes indicating higher frequencies. Dominant terms like "assign," "work," "office," and "class" suggest a focus on professional and academic tasks. The prominence of words like "quiet," "peace," and "room" may imply a preference for tranquil, reflective, or individual activities. The visualization showcases recurring themes and priorities.

Frequency Analysis of Top Words in Event Descriptions

[ ]:
```
from collections import Counter

# Combine all reviews into one corpus
corpus = ' '.join(df['events_descriptions_processed'])

# Tokenize the corpus
tokens = corpus.split()

# Create a frequency distribution
freq_dist = Counter(tokens)

# Convert to DataFrame for easier plotting
freq_df = pd.DataFrame(freq_dist.items(), columns=['Word', 'Frequency']).
 ↪sort_values(by="Frequency", ascending=False)

# Select top 20 words
top_words = freq_df.head(20)

# Plot the frequencies
plt.figure(figsize=(10, 8))
plt.bar(top_words['Word'], top_words['Frequency'])
plt.xticks(rotation=90)
plt.xlabel('Words')
plt.ylabel('Frequency')
plt.title('Top 20 Most Frequent Words')
plt.show()
```

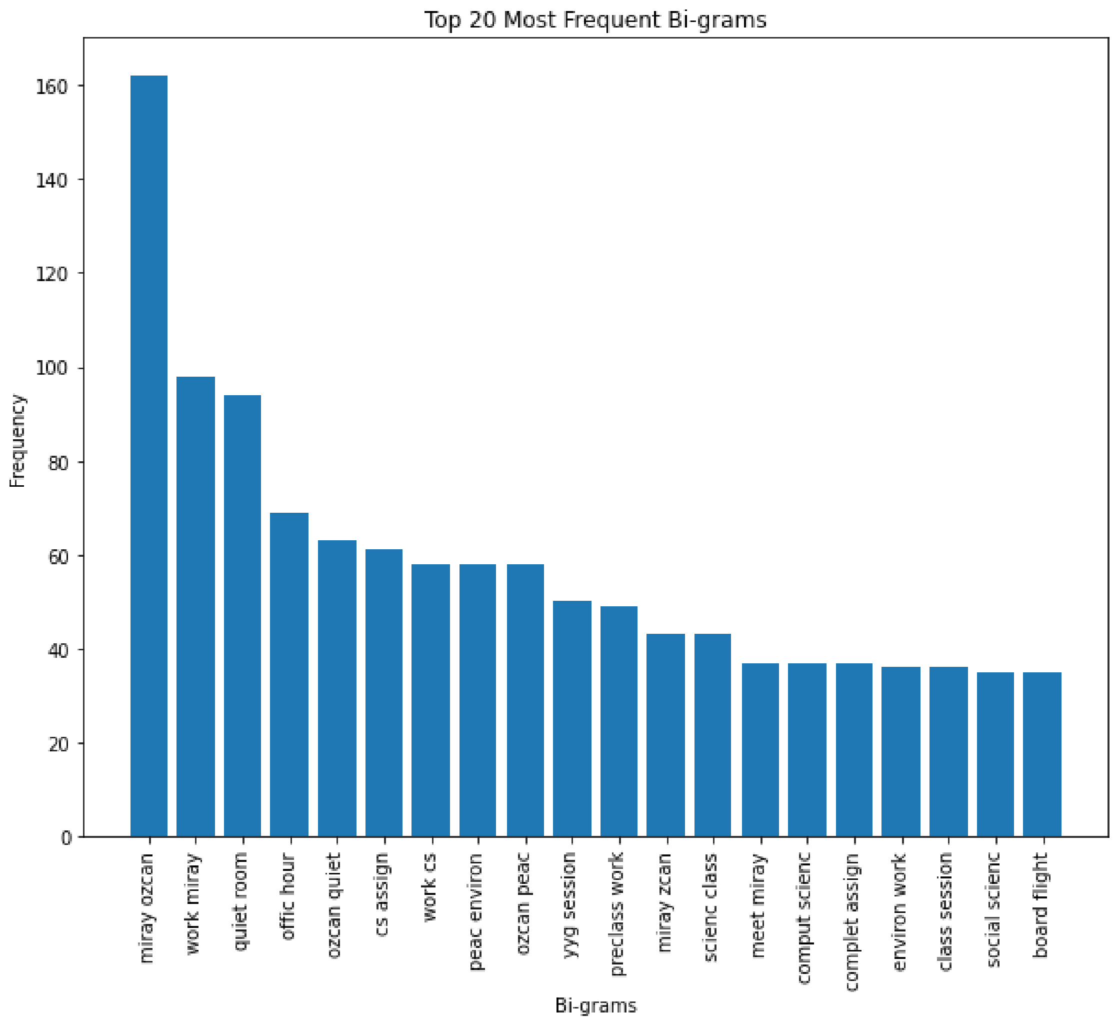


The bar chart showcases the most common words across the event descriptions, revealing a high occurrence of terms like "work," "session," "Miray," and "attend." The data indicates a significant emphasis on work-related activities and participation in sessions or meetings. The presence of a personal name suggests individual assignments or personalized tasks. This frequency analysis helps identify key areas of focus and recurring activities, providing insights into my professional and personal priorities as reflected in my event description data.

Bi-gram Frequency Analysis in Event Descriptions

```
[ ]: from sklearn.feature_extraction.text import CountVectorizer

# Create a bi-gram count vectorizer
bigram_vectorizer = CountVectorizer(ngram_range=(2, 2), max_features=1000)
bigram_matrix = bigram_vectorizer.
  ↪fit_transform(df['events_descriptions_processed'])

# Get feature names and sum up feature count
bigram_counts = bigram_matrix.sum(axis=0).A1
```

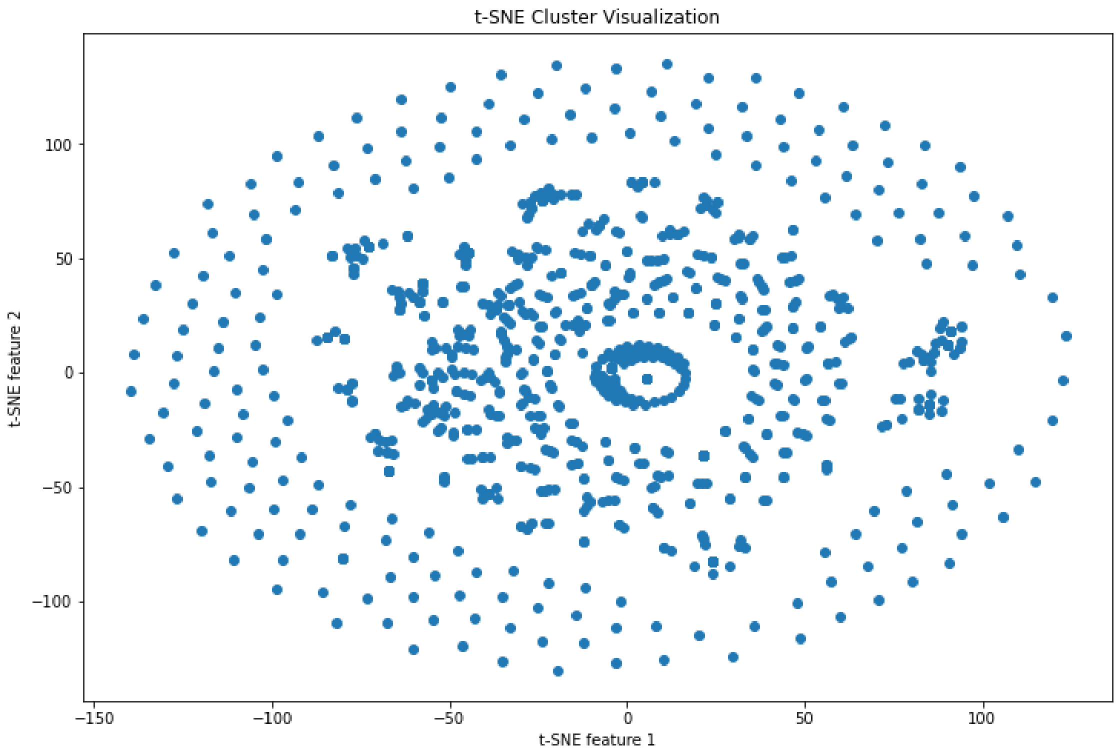
```
bigram_features = bigram_vectorizer.get_feature_names_out()  # Updated method␣
 ↪call
bigram_freq_df = pd.DataFrame({'Bi-gram': bigram_features, 'Frequency':␣
 ↪bigram_counts})

# Plot bi-gram frequencies
top_bigrams = bigram_freq_df.sort_values(by='Frequency', ascending=False).
 ↪head(20)
plt.figure(figsize=(10, 8))
plt.bar(top_bigrams['Bi-gram'], top_bigrams['Frequency'])
plt.xticks(rotation=90)
plt.xlabel('Bi-grams')
plt.ylabel('Frequency')
plt.title('Top 20 Most Frequent Bi-grams')
plt.show()
```



This bar chart visualizes the most common bi-gram combinations in the event descriptions, with pairs such as “Miray Ozcan,” “work session,” and “quiet room” leading the frequency count. The data suggests a focus on collaborative work, with recurring themes of planning and quiet environments for concentration. The frequent appearance of a personal name alongside key activities indicates individual involvement in various tasks or projects. The prominence of terms like “cs session” and “science class” reflects a strong academic or technical aspect in my routine or interests.

Text Data Clustering with t-SNE

[ ]:
```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.manifold import TSNE

# Convert text data into TF-IDF
vectorizer = TfidfVectorizer(max_features=1000)
X = vectorizer.fit_transform(df['events_descriptions_processed']).toarray()

# Initialize t-SNE
tsne = TSNE(n_components=2, random_state=0)

# Fit and transform t-SNE on the TF-IDF data
X_tsne = tsne.fit_transform(X)

# Plot the t-SNE results
plt.figure(figsize=(12, 8))
plt.scatter(X_tsne[:, 0], X_tsne[:, 1])
plt.xlabel('t-SNE feature 1')
plt.ylabel('t-SNE feature 2')
plt.title('t-SNE Cluster Visualization')
plt.show()
```

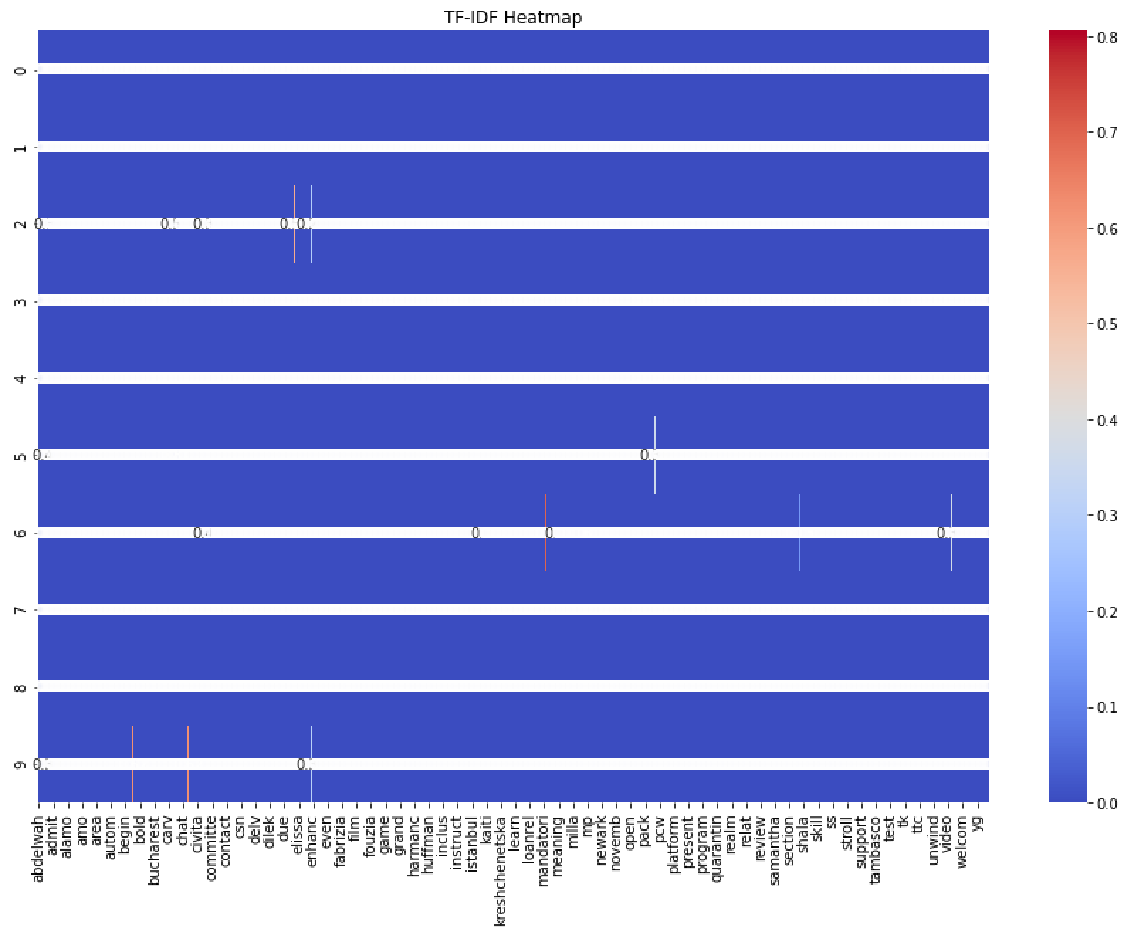


This t-SNE visualization presents a two-dimensional representation of the dataset's textual features, where each point corresponds to an event description. Despite some overlap, distinct clusters suggest patterns within the data, indicating that certain events are topically similar to each other. The tightness of clusters reflects the similarity in the content of event descriptions, whereas outliers might represent unique or less common topics. This visualization aids in understanding the natural groupings and diversity of the event topics, which can be leveraged for further analysis such as categorization or anomaly detection.

TF-IDF Feature Importance Heatmap

[ ]:
```
from sklearn.feature_extraction.text import TfidfVectorizer

# Initialize the vectorizer
vectorizer = TfidfVectorizer()

# Use the TF-IDF vectorizer to transform the text data
tfidf_matrix = vectorizer.fit_transform(df['events_descriptions_processed']).
 ↪toarray()

# Create a DataFrame for the TF-IDF data (take subset for visualization␣
 ↪purposes)
# Note the change from get_feature_names() to get_feature_names_out()
tfidf_df = pd.DataFrame(tfidf_matrix, columns=vectorizer.
 ↪get_feature_names_out()).head(10)
```

```python
# Create and plot a heatmap
plt.figure(figsize=(15, 10))
sns.heatmap(tfidf_df, annot=True, cmap='coolwarm')
plt.title('TF-IDF Heatmap')
plt.show()
```



This heatmap visualizes the significance of terms in a collection of event descriptions using TF-IDF, a statistical measure that evaluates how relevant a word is to a document in a collection of documents. The 'TF' stands for Term Frequency, while 'IDF' denotes Inverse Document Frequency. Darker colors indicate higher TF-IDF scores, meaning those terms are important in the context of their specific document but less common across all documents, highlighting their unique contribution to the topic. Lighter colors suggest the words are either commonly used across all documents or carry less importance in the given text. This analysis is essential for understanding key themes and differentiators within the event description data.

1.4 Section 4: Task Discussion and Data Splitting for Model Training (First and Second Pipeline)

1.4.1 Task Discussion and Its Evolution from the First Pipeline to Second Pipeline

In the **first pipeline**, the task at hand was a **classification** problem, where the objective was to predict the daily mood ('Relaxed,' 'Overwhelmed,' 'Busy,' 'Productive') based on various features extracted from calendar event data. The choice of classification was informed by the nature of the target variable, which is categorical with multiple classes.

Classification is a supervised learning approach that requires labeled data for training. In the first pipeline, the 'mood' column in the dataset served as the label, and the remaining columns with **numerical data** constituted the feature set. The model learned from this data to classify or predict the mood for a given day based on the input features.

In the **second pipeline**, the task is a **still classification** problem; however, this time, the objective is to predict the daily mood ('Negative,' 'Positive') based on various features extracted from the calendar event data, including the text data coming from event descriptions. Therefore, we will be training models on the mixture of numeric and text data (text data will be utilized after tokenization & vectorization).

1.4.2 Data Splitting Strategy

To evaluate the performance of the classification model, the combination of numerical data and tokenized & vectorized text data will be divided into three sets:

1. **Training Set:** Used to fit the model; it learns from this data.
2. **Validation Set:** Used to tune the model's hyperparameters and make decisions about the model, features, and preprocessing steps without touching the test set.
3. **Test Set:** Used to assess the performance of the model; it simulates the model's deployment on unseen data.

This splitting ensures that the model's performance is evaluated on data that it has never seen during the training phase, providing an unbiased estimate of its real-world performance.

```
[ ]: # Libraries for mixed (numerical + tokenized & vectorized text) data preparation

     from sklearn.feature_extraction.text import TfidfVectorizer
     from sklearn.model_selection import train_test_split
     from sklearn.preprocessing import StandardScaler
     from scipy.sparse import hstack
```

```
[ ]: # Load your dataset

     df = pd.read_csv('/Users/mirayozcan/Desktop/
      ↪daily_summary_chopped_labeled_enhanced.csv')

     # Ensure all text data is of string type
     df['events_descriptions_processed'] = df['events_descriptions_processed'].
      ↪fillna('')
```

```
# Proceed with the vectorization as before
vectorizer = TfidfVectorizer(max_features=1000)
X_text_tfidf = vectorizer.fit_transform(df['events_descriptions_processed'])


# Numeric features
numeric_features = ['total_duration', 'avg_duration', 'n_events',␣
 ↪'total_attendees',
                    'avg_attendees', 'start_first_event', 'end_last_event',
                    'day_of_month', 'day_of_week', 'has_event',␣
 ↪'segment_numeric']

X_numeric = df[numeric_features]

# Standardize the numeric features
scaler = StandardScaler()
X_numeric_scaled = scaler.fit_transform(X_numeric)

# Combine numeric and textual features
X = hstack([X_numeric_scaled, X_text_tfidf])

# Target variable
y = df['mood'].map({'Positive': 1, 'Negative': 0}).values


# Splitting the dataset into training (60%) and remaining (40%)
# The random_state parameter ensures reproducibility, meaning that every time␣
 ↪the code is run,
# the data is split the same way.
X_train, X_remaining, y_train, y_remaining = train_test_split(
    X, y, test_size=0.4, random_state=42
)

# Splitting the remaining data into validation and test sets (20% each of the␣
 ↪total data)
X_val, X_test, y_val, y_test = train_test_split(
    X_remaining, y_remaining, test_size=0.5, random_state=42
)

# Output the shapes of the resulting data splits to verify the sizes
print(f"Training features shape: {X_train.shape}")  # Expected: 60% of the data
print(f"Training labels shape: {y_train.shape}")
print(f"Validation features shape: {X_val.shape}")  # Expected: 20% of the data
print(f"Validation labels shape: {y_val.shape}")
print(f"Test features shape: {X_test.shape}")       # Expected: 20% of the data
print(f"Test labels shape: {y_test.shape}")
```

```
Training features shape: (914, 1011)
Training labels shape: (914,)
Validation features shape: (305, 1011)
Validation labels shape: (305,)
Test features shape: (305, 1011)
Test labels shape: (305,)
```

1.5 Section 5: Model Selection and Construction (Transitioning From First to Second Pipeline)

1.5.1 First Pipeline Recap

In the initial pipeline, Multinomial Logistic Regression (MLR) was chosen for its suitability for multi-class classification problems. The mood prediction task was approached as a classification of four distinct moods using features derived from calendar data. MLR's interpretability and its ability to handle both numerical and categorical data efficiently were key factors in its selection. The model was trained on a feature set consisting of derived numerical data from the events, and the 'mood' labels were encoded to fit the multi-class nature of the problem.

1.5.2 Transitioning to the Second Pipeline

The second pipeline brings several significant changes after updating the structure of the model training dataset and labels. The mood categories have been simplified into two: 'Positive' and 'Negative.' This alteration transforms the problem into a binary classification task. Additionally, text data from event descriptions is now incorporated, increasing the feature space and necessitating a revision in model selection.

While Logistic Regression remains a candidate due to its robustness and ease of interpretation, it's essential to explore other models that may offer improved performance given the changes in the dataset. Specifically, models that can accommodate the high-dimensional sparse data resulting from text vectorization need to be considered.

The incorporation of text data adds complexity to the feature space. TF-IDF vectorization converts text data into numerical form, capturing the importance of terms relative to the document and corpus. This results in a high-dimensional sparse matrix which Logistic Regression can handle but might not leverage as effectively as tree-based methods or support vector machines (SVM).

Brief Rationale for Including New Models:

- **Binary Logistic Regression:** Although the task has shifted from multi-class to binary classification, Logistic Regression remains a strong baseline model. Its coefficients offer insights into feature importance, but it may not capture complex relationships as effectively as other models.
- **XGBoost:** An implementation of gradient-boosted decision trees designed for speed and performance. XGBoost can handle the sparsity of the data and is capable of capturing non-linear relationships. It also includes regularized boosting to prevent overfitting.
- **SVM:** SVMs are powerful for classification problems, particularly when dealing with high-dimensional data. Their kernel trick can capture complex relationships between features without the need for explicit transformation, making them suitable for text classification tasks.

1.5.3 Model 1: Binary Logistic Regression (Transitioning from the First Pipeline)

1.1 Model Selection Discussion Transitioning from the first pipeline, where we utilized Multinomial Logistic Regression (MLR) for multi-class mood prediction, we now shift towards Binary Logistic Regression (BLR) for our updated binary classification task. This change aligns with the revised mood categories of 'Positive' and 'Negative', simplifying our target variable.

BLR is an appropriate choice for several reasons. Firstly, it's well-suited for binary classification problems like ours, offering a direct method to model the probability of one mood state over another. It maintains computational efficiency, crucial for handling our dataset that includes both numerical and text-based features.

The incorporation of event descriptions as text features introduces a high-dimensional space, which BLR can navigate efficiently. Through the use of techniques like TF-IDF vectorization for text data, we can maintain a manageable feature space without overwhelming the model.

Moreover, the interpretability of BLR, much like MLR, remains a significant advantage. The model coefficients indicate the impact of each feature on the probability of observing a 'Positive' or 'Negative' mood, which is valuable for understanding which aspects of the calendar data most influence mood predictions.

While BLR is robust and interpretable, it's essential to acknowledge its assumptions and limitations. It presumes a linear relationship between the log-odds of the outcome and the input features, which might not always hold true. Additionally, BLR assumes that features are independent of each other, an assumption that may not apply in all scenarios, especially with complex datasets.

Despite these limitations, BLR's benefits—particularly its efficiency, interpretability, and suitability for binary outcomes—make it a solid choice for our task. To address potential shortcomings and enhance model performance, we'll also explore other models like XGBoost and SVC, considering them alongside BLR to find the best approach for our mood prediction task.

1.2 Mathematical Underpinnings of the Model Binary Logistic Regression (BLR) is leveraged for predicting binary outcomes, such as pass/fail, win/lose, or yes/no scenarios. It is a special case of linear regression where the target variable is categorical in nature. Here's an overview of BLR and its mathematical foundations:

Model Formulation

The logistic function, also known as the sigmoid function, models the probability that the target variable Y belongs to a particular category. For a binary classification problem, this probability can be expressed as:

$$P(Y = 1|X = x) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 x_1 + \ldots + \beta_p x_p)}}$$

where: - Y is the binary dependent variable. - X denotes the independent variables. - $\beta_0, \beta_1, ..., \beta_p$ are the model coefficients.

Maximum Likelihood Estimation (MLE)

MLE is used to find the best-fitting model to describe the relationship between the dependent variable and a set of independent variables. It does this by maximizing a likelihood function, which measures how well the model predicts the observed data. The likelihood for all observations is:

$$L(\beta) = \prod_{i=1}^{n} P(y_i|x_i)^{y_i}(1 - P(y_i|x_i))^{1-y_i}$$

The goal is to maximize this likelihood function, which, for computational reasons, is equivalent to minimizing the negative log-likelihood:

$$\ell(\beta) = -\sum_{i=1}^{n} [y_i \log(P(y_i|x_i)) + (1 - y_i)\log(1 - P(y_i|x_i))]$$

Gradient Descent

Gradient descent is an optimization algorithm used to minimize the loss function in logistic regression, which is typically the negative log-likelihood. The gradient of the loss function with respect to the coefficients is computed and used to update the coefficients in the direction that reduces the loss:

$$\beta := \beta - \alpha \nabla_\beta \ell(\beta)$$

where α is the learning rate.

Regularization

Regularization adds a penalty to the loss function to control overfitting. L2 regularization (Ridge) is commonly applied in logistic regression:

$$\ell_{\text{regularized}}(\beta) = \ell(\beta) + \lambda \sum_{j=1}^{p} \beta_j^2$$

where λ is the regularization strength.

Interpretation of Coefficients

In logistic regression, the coefficients represent the log odds of the outcome. For a one-unit increase in x_i, the expected change in log odds is β_i.

Handling of Numerical and Categorical Data

Numerical predictors are used directly, while categorical predictors are typically one-hot encoded before model fitting.

By adhering to these principles, BLR provides a robust framework for binary classification tasks.

1.3 Model Initialization and Construction In this updated pipeline, we initialize and construct a binary logistic regression model using the scikit-learn pipeline module. The pipeline still incorporates two principal components: the StandardScaler for feature scaling and the LogisticRegression model. Unlike the previous pipeline where the logistic regression was configured for multi-class classification with the 'multinomial' option, we now use the default binary setting that fits our updated label structure of 'Positive' and 'Negative' mood predictions. The 'lbfgs' solver remains the optimization algorithm of choice, given its efficacy in handling binary logistic regression's loss function. This approach ensures that our model is tailored to the nuances of binary classification while benefiting from the robustness and computational efficiency of logistic regression.

```
[ ]: # Libraries

from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
from sklearn.metrics import accuracy_score, classification_report

# Initialize the model
logistic_model = LogisticRegression(max_iter=1000)

# Define a pipeline that includes scaling and logistic regression
pipeline = Pipeline([
    ('scaler', StandardScaler(with_mean=False)),
    ('classifier', logistic_model)
])
```

1.5.4 Model 2: XGBoost (New for the Second Pipeline)

2.1 Model Selection Discussion In the evolution of my dataset, I've chosen to integrate XGBoost, a sophisticated ensemble algorithm, as a natural progression from simpler models previously considered. The shift towards XGBoost is informed by its proficiency in managing the complexity introduced by my enhanced dataset, which combines quantitative insights with qualitative text analysis to predict mood states.

XGBoost, standing for Extreme Gradient Boosting, builds on the premise of gradient boosting machines but with a performance-optimized approach suitable for our project's needs. It shines in scenarios where both the volume and variety of data necessitate a robust, scalable solution. The algorithm excels at handling the vast, sparse matrices resulting from text vectorization and thrives amidst the myriad of numeric features my calendar data provides.

The algorithm's use of gradient boosting is pivotal. By sequentially correcting the mistakes of previous models, XGBoost ensures incremental improvements that are essential for the nuanced prediction of mood states. It's particularly adept at navigating the subtleties within our textual data, refining its predictive acuity with each iteration.

XGBoost's regularization techniques address the overfitting risks, a critical measure considering our model's exposure to a diverse feature set. The ability to automatically handle missing data further aligns with the erratic nature of real-world data collection, as seen in our project's dataset.

Despite its strengths, XGBoost presents a complexity that may obscure the 'why' behind its predictions. This trade-off, between predictive power and model transparency, is a considered choice. My project prioritizes accuracy and performance in mood prediction, thereby validating the choice of XGBoost despite the potential interpretability compromise.

2.2 Mathematical Underpinnings of the Model **Model Formulation**

XGBoost stands for Extreme Gradient Boosting, an advanced implementation of gradient boosting that is both efficient and effective for a wide range of data science problems, including our task of mood prediction from complex datasets. It's built upon the principles of boosting, particularly gradient boosting machines, but with significant improvements in speed and performance.

Ensemble Learning and Boosting

XGBoost belongs to the family of ensemble learning methods, which improves prediction accuracy by combining the strengths of a collection of simpler base models. Boosting, a form of ensemble learning, incrementally builds an ensemble by training each new model instance to emphasize the data points that previous models misclassified.

Gradient Boosting Framework

The core of XGBoost lies in its use of the gradient boosting framework. In gradient boosting, base learners are added one at a time, and each new base learner is trained to correct the errors made by the combined ensemble of all previous learners. The ensemble model's prediction $\hat{y}_i$ for an observation x_i after k iterations is:

$$\hat{y}_i^{(k)} = \sum_{j=1}^{k} f_j(x_i)$$

where f_j represents the prediction of the j-th base learner.

Objective Function

The objective function that XGBoost aims to minimize comprises a loss term and a regularization term:

$$\text{Obj}^{(k)} = \sum_{i=1}^{n} \ell(\hat{y}_i, y_i) + \sum_{j=1}^{k} \Omega(f_j)$$

Here, ℓ is a differentiable convex loss function that measures the difference between the predicted $\hat{y}_i$ and the actual y_i, and Ω is the regularization term which penalizes the complexity of the model.

Regularization

XGBoost's regularization term helps prevent overfitting, which is crucial in the context of our feature-rich text and numerical data. The regularization term $\Omega(f)$ for a tree f is defined as:

$$\Omega(f) = \gamma T + \frac{1}{2}\lambda \sum_{j} w_j^2$$

where T is the number of leaves in the tree, w_j is the score on the j-th leaf, γ is the penalty for each additional leaf, and λ is the L2 regularization term on the leaf weights.

Gradient and Hessian

XGBoost uses a second-order approximation to optimize the objective function. The Taylor expansion of the loss function up to the second order is used, which involves both the gradient and Hessian (second derivative of the loss function):

$$\ell(y, \hat{y}) \approx \ell(y, \hat{y}^{(t-1)}) + g_i f_t(x_i) + \frac{1}{2} h_i f_t(x_i)^2$$

where g_i and h_i are the first and second order partial derivatives of the loss function with respect to $\hat{y}_i^{(t-1)}$ (the prediction of the i-th instance at the $(t-1)$-th iteration), respectively.

Tree Pruning and Split Finding

XGBoost improves upon the traditional gradient boosting decision tree algorithm with a more regularized model formalization to control overfitting, which makes it an effective algorithm for a wide range of data science tasks, including this mood prediction challenge. It utilizes a novel sparsity-aware algorithm for parallel tree learning and an approximate greedy algorithm for split finding to handle different types of sparsity patterns in the input data effectively.

2.3 Model Initialization and Construction For this project, we initialize and construct an XGBoost model using the `xgboost` package. XGBoost's popularity in machine learning competitions and industry applications is due to its performance and speed, particularly when handling large datasets with complex features.

```
[ ]: # Libraries

import xgboost as xgb
from sklearn.metrics import accuracy_score, classification_report

# Initialize the XGBoost classifier
xgboost_model = xgb.XGBClassifier(objective='binary:logistic',␣
 ↪eval_metric='logloss')
```

1.5.5 Model 3: Support Vector Machine (New for the Second Pipeline)

3.1 Model Selection Discussion In advancing my project, I've pivoted towards incorporating Support Vector Machines (SVM) into my suite of models to predict mood from calendar data. This strategic selection stems from SVM's unparalleled capacity to decipher the intricacies of high-dimensional spaces, a characteristic feature of my augmented dataset that interlaces dense textual information with numerical insights.

SVM operates on the principle of finding the optimal hyperplane that offers the maximum margin between different mood states. This methodological approach is especially pertinent to my project, where the distinction between 'Positive' and 'Negative' moods hinges on subtle variations across a rich feature space. The application of kernel tricks enables SVM to transform the feature space into higher dimensions where these subtle distinctions become linearly separable, enhancing the model's discriminatory power.

One of the compelling attributes of SVM for this project is its versatility in handling both linear and non-linear relationships, facilitated by the selection of appropriate kernel functions. This adaptability is crucial in unraveling complex patterns within the textual data, potentially unlocking more profound insights into mood prediction dynamics.

While SVM's methodical approach to maximizing classification margins bolsters its prediction accuracy, it also introduces challenges related to model tuning and computational efficiency. The need to meticulously select the kernel function and tune hyperparameters like C (regularization parameter) and gamma (kernel coefficient) demands a thoughtful consideration, given the computational resources at hand.

3.2 Mathematical Underpinnings of the Model   Model Formulation

Support Vector Machine (SVM) constructs a hyperplane in a high or infinite-dimensional space, which can be used for classification, regression, or other tasks. Intuitively, a good separation is achieved by the hyperplane that has the largest distance to the nearest training data point of any class (so-called functional margin), since in general the larger the margin, the lower the generalization error of the classifier.

For a linear SVM classifier, the decision function is defined as:

$$f(x) = \mathbf{w}^T \mathbf{x} + b$$

where: - $\mathbf{w}$ is the weight vector. - $\mathbf{x}$ is the input features. - b is the bias.

The goal is to find the values of $\mathbf{w}$ and b that maximize the margin between the two classes. The optimization problem can be formulated as:

$$\min_{\mathbf{w},b} \frac{1}{2}\|\mathbf{w}\|^2$$

subject to $(y_i(\mathbf{w}^T\mathbf{x}_i + b) \geq 1)$ for $i = 1, \ldots, n$, where $y_i \in \{-1, 1\}$ are the labels of the training examples.

Kernel Trick

The kernel trick involves transforming the data into another dimension that has a clear dividing margin between classes of data. In the case of non-linearly separable data, SVM uses a kernel function to map the inputs into high-dimensional feature spaces where a linear separation is possible.

A commonly used kernel function is the Radial Basis Function (RBF):

$$K(\mathbf{x}_i, \mathbf{x}_j) = \exp(-\gamma\|\mathbf{x}_i - \mathbf{x}_j\|^2)$$

where γ is a parameter that sets the 'spread' of the kernel.

Support Vectors

Support vectors are the data points that lie closest to the decision surface (or hyperplane). They are critical to defining the hyperplane because the orientation and position of the hyperplane are determined by these data points. Essentially, the support vectors are the most difficult to classify and give the most information about the decision boundary.

Regularization

Regularization is crucial in SVM to prevent overfitting. The regularization parameter (often denoted as C) tells the SVM optimization how much you want to avoid misclassifying each training example. For large values of C, the optimization will choose a smaller-margin hyperplane if that hyperplane does a better job of getting all the training points classified correctly.

3.3 Model Initialization and Construction Recognizing the nuanced intricacies of our dataset—spanning both the rich, textual descriptions of calendar events and numerous quantitative event features—SVC stands out for its robustness and versatility in handling high-dimensional spaces. This model is particularly apt for our binary classification task, aiming to distinguish between 'Positive' and 'Negative' mood outcomes based on the intricate interplay of our feature set.

The construction of the SVC model within our pipeline follows a structured approach, incorporating both preprocessing and model training stages for optimal performance. Below is how we integrate SVC into our data processing pipeline, leveraging scikit-learn's comprehensive suite of tools.

[ ]:
```
# Libraries

from sklearn.svm import SVC

# Initialize the Support Vector Classifier with tuned hyperparameters
svc_model = SVC(kernel='rbf', C=1.0, gamma='scale')

# Construct a pipeline that integrates feature scaling with SVC
pipeline_svc = Pipeline([
    ('scaler', StandardScaler(with_mean=False)),  # Normalize feature vectors␣
 ↪for improved classification performance
    ('svc', svc_model)              # Support Vector Classifier to model the␣
 ↪mood prediction task
])
```

1.6 Section 6: Models' Training, Cross-Validation, and Hyperparameters' Tuning (Second Pipeline)

In this section, we train the Binary Logistic Regression, XGBoost, and SVM models, perform cross-validation, and conduct hyperparameter tuning for each of the models to optimize their performances.

1.6.1 Model 1: Binary Logistic Regression

Model Training and Cross-Validation For Binary Logistic Regression, we'll employ scikit-learn's GridSearchCV to automate the search for the optimal hyperparameters while incorporating cross-validation. This methodology ensures that our model is not just fitted to a specific subset of the data, thus enhancing its generalization capabilities.

First, we establish a grid of potential hyperparameters. For Logistic Regression, we focus on the regularization strength (C) and the type of solver used. This allows us to balance model complexity with training efficiency, critical for managing the high-dimensional space of our features.

[ ]:
```
from sklearn.model_selection import GridSearchCV

# Hyperparameter grid
param_grid = {
    'classifier__C': [0.01, 0.1, 1, 10],  # Exploration of regularization␣
 ↪strengths
    'classifier__solver': ['lbfgs', 'liblinear']  # Solver options
}
```

We integrate GridSearchCV with our pipeline, specifying the hyperparameter grid, the number of folds for cross-validation, and the scoring metric. This approach automates the iterative training and validation process across different hyperparameter combinations.

[ ]:
```
# Initialize GridSearchCV
grid_search = GridSearchCV(estimator=pipeline, param_grid=param_grid, cv=5,␣
 ↪scoring='accuracy')

# Fit GridSearchCV to the training data
grid_search.fit(X_train, y_train)
```

[ ]:
```
GridSearchCV(cv=5,
             estimator=Pipeline(steps=[('scaler',
                                        StandardScaler(with_mean=False)),
                                       ('classifier',
                                        LogisticRegression(max_iter=1000))]),
             param_grid={'classifier__C': [0.01, 0.1, 1, 10],
                         'classifier__solver': ['lbfgs', 'liblinear']},
             scoring='accuracy')
```

Post-training, we extract the best hyperparameter combination identified through GridSearchCV. This step is pivotal in fine-tuning our model's configuration to our dataset's specifics.

[ ]:
```
# Retrieve the best hyperparameters
best_params_lr = grid_search.best_params_
print(f"Optimized hyperparameters: {best_params_lr}")
```

```
Optimized hyperparameters: {'classifier__C': 0.01, 'classifier__solver':
'lbfgs'}
```

Hyperparameter Tuning and Model Evaluation With the optimal hyperparameters at hand, we evaluate the model's performance on a separate validation set. This step is instrumental in assessing how well our model generalizes to unseen data.

[ ]:
```
# Evaluate the optimized model
best_model_lr = grid_search.best_estimator_
y_val_pred_lr = best_model_lr.predict(X_val)

# Generate and print the classification report
from sklearn.metrics import classification_report

print(classification_report(y_val, y_val_pred_lr))
```

```
              precision    recall  f1-score   support

           0       0.82      0.51      0.63       172
           1       0.58      0.86      0.69       133

    accuracy                           0.66       305
   macro avg       0.70      0.68      0.66       305
weighted avg       0.71      0.66      0.66       305
```

The updated classification report for the Binary Logistic Regression model provides insight into its performance on the validation set, distinguished between 'Negative' (0) and 'Positive' (1) mood states:

- **Precision:** The model predicts 'Negative' moods with a precision of 0.82, indicating a strong accuracy for this class. In contrast, its precision for 'Positive' moods is 0.58, implying that there are more false positives for this category.
- **Recall:** The recall score for 'Negative' moods is lower at 0.51, showing that the model is missing almost half of the actual 'Negative' instances. For 'Positive' moods, the recall is quite high at 0.86, suggesting that the model is adept at identifying 'Positive' mood instances.
- **F1-Score:** The F1-scores, which balance precision and recall, stand at 0.63 for 'Negative' moods and 0.69 for 'Positive' moods. This indicates that the model is slightly better at balancing precision and recall for 'Positive' moods.
- **Support:** The support numbers, indicating the true occurrences of each class in the validation set, show 172 instances of 'Negative' moods and 133 of 'Positive' moods.
- **Accuracy:** The overall model accuracy is 0.66, indicating it correctly predicts the mood 66% of the time across both classes.
- **Macro Avg:** The macro average scores for precision, recall, and F1-score are all 0.70 for precision, 0.68 for recall, and 0.66 for the F1-score, giving equal weight to both classes irrespective of their prevalence.
- **Weighted Avg:** The weighted average scores for precision, recall, and F1-score are 0.71, 0.66, and 0.66, respectively. These averages consider the imbalance in class distribution, suggesting a slightly better precision across the classes.

Overall, the Binary Logistic Regression model is more precise when predicting 'Negative' moods but less so for 'Positive' moods. While it effectively identifies most 'Positive' mood instances, it tends to miss a significant portion of 'Negative' moods. Efforts to improve the model should focus on increasing its ability to recognize 'Negative' moods without compromising the high recall for 'Positive' moods.

1.6.2 Model 2: XGBoost

Model Training and Cross-Validation XGBoost stands out for its efficiency and effectiveness in dealing with large and complex datasets. To optimize the performance of the XGBoost model, we will apply cross-validation and hyperparameter tuning using GridSearchCV from scikit-learn.

Cross-validation ensures our model's robustness by evaluating its performance on different data subsets, thereby providing a comprehensive assessment of its generalization capability.

```
[ ]: from sklearn.model_selection import GridSearchCV
     import xgboost as xgb
     from sklearn.metrics import classification_report

     # Define hyperparameters to search over
     param_grid = {
         'max_depth': [3, 4, 5],
         'learning_rate': [0.01, 0.1, 0.3],
         'n_estimators': [100, 200, 300],
         'subsample': [0.8, 0.9, 1]
     }

     # Initialize GridSearchCV with the XGBoost classifier and hyperparameter grid
     grid_search = GridSearchCV(estimator=xgboost_model, param_grid=param_grid,␣
      ↪cv=5, scoring='accuracy')

     # Fit GridSearchCV to the training data
     grid_search.fit(X_train, y_train)
```

```
[ ]: GridSearchCV(cv=5,
                  estimator=XGBClassifier(base_score=None, booster=None,
                                          callbacks=None, colsample_bylevel=None,
                                          colsample_bynode=None,
                                          colsample_bytree=None, device=None,
                                          early_stopping_rounds=None,
                                          enable_categorical=False,
                                          eval_metric='logloss', feature_types=None,
                                          gamma=None, grow_policy=None,
                                          importance_type=None,
                                          interaction_constraints=None,
                                          learning_rate=…
                                          max_cat_to_onehot=None,
                                          max_delta_step=None, max_depth=None,
                                          max_leaves=None, min_child_weight=None,
                                          missing=nan, monotone_constraints=None,
                                          multi_strategy=None, n_estimators=None,
                                          n_jobs=None, num_parallel_tree=None,
                                          random_state=None, …),
                  param_grid={'learning_rate': [0.01, 0.1, 0.3],
                              'max_depth': [3, 4, 5],
                              'n_estimators': [100, 200, 300],
                              'subsample': [0.8, 0.9, 1]},
                  scoring='accuracy')
```

```
[ ]: # Retrieve the best hyperparameters
     best_params_xg = grid_search.best_params_
     print(f"Optimized hyperparameters: {best_params_xg}")
```

```
Optimized hyperparameters: {'learning_rate': 0.01, 'max_depth': 4,
'n_estimators': 200, 'subsample': 0.8}
```

Hyperparameter Tuning and Model Evaluation

```
[ ]: # Evaluate the optimized model
     best_xgboost_model = grid_search.best_estimator_
     y_val_pred_xg = best_xgboost_model.predict(X_val)

     # Generate and print the classification report
     print(classification_report(y_val, y_val_pred_xg))
```

```
              precision    recall  f1-score   support

           0       0.87      0.52      0.65       172
           1       0.59      0.90      0.71       133

    accuracy                           0.69       305
   macro avg       0.73      0.71      0.68       305
weighted avg       0.75      0.69      0.68       305
```

The updated classification report for the XGBoost model elucidates its performance on the validation set, identified as ‘Negative’ (0) and ‘Positive’ (1) mood predictions:

- **Precision:** The model exhibits a high precision of 0.87 for the ‘Negative’ class, indicating a strong propensity for accurate negative predictions. For the ‘Positive’ class, precision stands at 0.59, suggesting a relatively higher incidence of false positives when predicting positive moods.
- **Recall:** The model demonstrates a recall of 0.52 for the ‘Negative’ class, meaning it correctly identifies just over half of the actual negative instances. However, it performs notably better in detecting ‘Positive’ moods with a recall of 0.90.
- **F1-Score:** The F1-score, which factors both precision and recall, is 0.65 for ‘Negative’ and slightly higher at 0.71 for ‘Positive’ moods, indicating a more balanced performance for the latter.
- **Support:** There are 172 instances of ‘Negative’ moods and 133 of ‘Positive’ moods within the validation set, denoted by the support numbers.
- **Accuracy:** The overall accuracy is computed at 0.69, showcasing that the model accurately predicts mood states nearly 69% of the time across both categories.
- **Macro Avg:** The macro averages for precision, recall, and F1-score are calculated as 0.73, 0.71, and 0.68, respectively. These values average the performance across both classes without regard to their prevalence.

- **Weighted Avg:** The weighted averages incorporate the class distribution and stand at 0.75 for precision, 0.69 for recall, and 0.68 for the F1-score. These figures offer a quality measure reflective of each class's occurrence.

Overall, the XGBoost model is adept at predicting 'Positive' moods, as evidenced by the high recall rate, but there is a tendency to overpredict this class, given the lower precision. 'Negative' mood predictions are accurate when they occur, but the model frequently misses these instances, as suggested by the recall. Optimizing the model may require a focus on improving the detection of 'Negative' moods without losing the high recall for 'Positive' moods.

1.6.3 Model 3: Support Vector Machine

Model Training and Cross-Validation For Support Vector Machines, we'll utilize scikit-learn's GridSearchCV to conduct a thorough search for the best hyperparameters while employing cross-validation to assess the model's robustness and generalization capability.

```
[ ]: from sklearn.model_selection import GridSearchCV
     from sklearn.metrics import classification_report

     # Define hyperparameters to search over for SVM
     param_grid_svc = {
         'svc__C': [0.1, 1, 10],  # Regularization parameter
         'svc__gamma': ['scale', 'auto'],  # Kernel coefficient
         'svc__kernel': ['rbf', 'linear']  # Specifies the kernel type
     }

     # Initialize GridSearchCV with the SVM classifier and hyperparameter grid
     grid_search_svc = GridSearchCV(estimator=pipeline_svc,␣
      ↪param_grid=param_grid_svc, cv=5, scoring='accuracy')

     # Fit GridSearchCV to the training data
     grid_search_svc.fit(X_train, y_train)
```

```
[ ]: GridSearchCV(cv=5,
                  estimator=Pipeline(steps=[('scaler',
                                             StandardScaler(with_mean=False)),
                                            ('svc', SVC())]),
                  param_grid={'svc__C': [0.1, 1, 10],
                              'svc__gamma': ['scale', 'auto'],
                              'svc__kernel': ['rbf', 'linear']},
                  scoring='accuracy')
```

```
[ ]: # Retrieve the best hyperparameters
     best_params_svc = grid_search_svc.best_params_
     print(f"Optimized hyperparameters for SVC: {best_params_svc}")
```

```
Optimized hyperparameters for SVC: {'svc__C': 1, 'svc__gamma': 'auto',
'svc__kernel': 'rbf'}
```

Hyperparameter Tuning and Model Evaluation

[ ]:
```python
# Evaluate the optimized SVM model
best_svc_model = grid_search_svc.best_estimator_
y_val_pred_svc = best_svc_model.predict(X_val)

# Generate and print the classification report for the validation set
print(classification_report(y_val, y_val_pred_svc))
```

```
              precision    recall  f1-score   support

           0       0.79      0.53      0.64       172
           1       0.58      0.82      0.68       133

    accuracy                           0.66       305
   macro avg       0.68      0.68      0.66       305
weighted avg       0.70      0.66      0.66       305
```

The classification report for the Support Vector Machine model gives us insights into its validation set performance across two mood prediction classes:

- **Precision**: The model achieves a precision of 0.79 for the 'Negative' class (0), indicating it is correct in its predictions 79% of the time. For the 'Positive' class (1), the precision is lower at 0.58, which suggests a higher chance of false positives for this class.
- **Recall**: The recall for the 'Negative' class stands at 0.53, showing that the model identifies a little over half of the actual negative instances. However, it shows a better recall of 0.82 for the 'Positive' class, indicating a strong ability to capture the majority of positive mood instances.
- **F1-Score**: For 'Negative' moods, the F1-score is 0.64, while for 'Positive' moods, it is higher at 0.68, indicating a slightly better balance between precision and recall for the positive predictions.
- **Support**: The support number shows 172 instances for 'Negative' and 133 for 'Positive', highlighting the number of actual occurrences of each mood class in the validation dataset.
- **Accuracy**: The overall model accuracy is calculated to be 0.66, meaning it correctly predicts mood classes 66% of the time when considering both classes together.
- **Macro Avg**: The macro average figures for precision, recall, and the F1-score are all equal at 0.68, suggesting a balanced performance across both mood classes without taking their distribution into account.
- **Weighted Avg**: The weighted average, which considers the class distribution, stands at 0.70 for precision and 0.66 for both recall and the F1-score. These weighted figures indicate that the model's performance is slightly better for the 'Negative' class due to its higher prevalence.

While the SVM model performs relatively well, it shows a notable discrepancy between the classes, with stronger precision for 'Negative' mood predictions but lower recall, suggesting missed negative instances. Conversely, the model tends to identify positive instances more frequently but with less precision. Improvements may involve refining the model to enhance its capacity to distinguish between moods, aiming for a better balance in precision and recall for both classes.

1.7 Section 7: Predictions for Out of Sample Data and Performance Evaluation for Models (Second Pipeline)

In this section, we generate predictions for out-of-sample data (the test set) using the best models obtained from the grid search process for binary logistic regression, XGBoost, and SVM models. Subsequently, we compute various performance metrics to evaluate each model's performance on the test set.

1.7.1 Model 1: Binary Logistic Regression

Generating Predictions With the optimized model from the hyperparameter tuning process, we now proceed to apply this model to predict the mood labels on the test set. This will provide us with an insight into how well the BLR model generalizes to unseen data.

```
[ ]: # Generate predictions for the test set using the best logistic regression model
     y_test_pred_lr = best_model_lr.predict(X_test)
```

Computing Performance Metrics

```
[ ]: # Import necessary functions from sklearn.metrics
     from sklearn.metrics import accuracy_score, precision_recall_fscore_support,␣
      ↪classification_report

     # Compute accuracy
     test_accuracy_lr = accuracy_score(y_test, y_test_pred_lr)
     print(f"Test Accuracy: {test_accuracy_lr}")

     # Generate a classification report
     test_classification_report_lr = classification_report(y_test, y_test_pred_lr)
     print("Classification Report on Test Set:\n", test_classification_report_lr)
```

```
Test Accuracy: 0.6786885245901639
Classification Report on Test Set:
               precision    recall  f1-score   support

           0       0.75      0.54      0.63       154
           1       0.64      0.82      0.72       151

    accuracy                           0.68       305
   macro avg       0.70      0.68      0.67       305
weighted avg       0.70      0.68      0.67       305
```

Interpretation of Performance Metrics The Binary Logistic Regression model displays a test accuracy of approximately 67.87%. This indicates that around two-thirds of the mood predictions made by the model on the test dataset are accurate.

The performance metrics for each class are as follows:

- Class 0 ('Negative' mood):
  - **Precision**: At 0.75, the model's predictions are correct three-quarters of the time when it predicts a negative mood.
  - **Recall**: The recall of 0.54 indicates that the model identifies just over half of the actual negative mood instances.
- Class 1 ('Positive' mood):
  - **Precision**: With a precision of 0.64, when the model predicts a positive mood, it is correct approximately 64% of the time.
  - **Recall**: The model exhibits a robust recall of 0.82, suggesting it is able to recognize the vast majority of positive mood instances.
- **F1-Scores**: The F1-scores, which harmonize precision and recall, are 0.63 for negative moods and 0.72 for positive moods. The model is slightly more effective at balancing precision and recall when predicting positive moods.
- **Support**: The support numbers, which indicate the actual occurrences of each class in the test set, are relatively balanced, with 154 instances of negative moods and 151 of positive moods.
- **Macro Average**: The macro averages are uniform across precision, recall, and the F1-score, all at 0.68. This metric suggests a balanced average performance for both classes without bias towards either class's prevalence.
- **Weighted Average**: The weighted averages, which consider the number of instances of each class, are also consistent at 0.70 for precision and 0.68 for both recall and the F1-score. These weighted metrics reflect a performance that takes into account the slight variation in class distribution within the test dataset.

Overall, while the model has a reasonable level of accuracy overall, it shows better precision for negative moods but struggles with their recall, indicating missed negative instances. Conversely, it has lower precision for positive moods but higher recall, suggesting it more reliably identifies positive instances. This indicates potential areas for improvement, particularly in enhancing the detection of negative moods and reducing false positives for positive mood predictions.

1.7.2 Model 2: XGBoost

Generating Predictions With the optimized model from the hyperparameter tuning process, we now proceed to apply this model to predict the mood labels on the test set. This will provide us with an insight into how well the XGBoost model generalizes to unseen data.

```
[ ]: # Generate predictions for the test set using the best XGBoost model
     y_test_pred_xg = best_xgboost_model.predict(X_test)
```

Computing Performance Metrics

```
[ ]: # Compute accuracy
     test_accuracy_xg = accuracy_score(y_test, y_test_pred_xg)
     print(f"Test Accuracy: {test_accuracy_xg}")
```

```python
# Generate a classification report
test_classification_report_xg = classification_report(y_test, y_test_pred_xg)
print("Classification Report on Test Set:\n", test_classification_report_xg)
```

```
Test Accuracy: 0.6852459016393443
Classification Report on Test Set:
               precision    recall  f1-score   support

           0       0.77      0.54      0.63       154
           1       0.64      0.83      0.72       151

    accuracy                           0.69       305
   macro avg       0.70      0.69      0.68       305
weighted avg       0.70      0.69      0.68       305
```

Interpretation of Performance Metrics The XGBoost model demonstrates a test accuracy of approximately 68.52%. This accuracy reflects the proportion of total predictions that were correct across the test dataset.

- Class 0 ('Negative' mood state):
  - **Precision**: At 0.77, this suggests that the model's predictions are correct about 77% of the time when it classifies a mood as 'Negative'.
  - **Recall**: The model identifies 54% of the actual 'Negative' instances, indicating that nearly half of the 'Negative' moods are being missed.
- Class 1 ('Positive' mood state):
  - **Precision**: A precision of 0.64 means that when the model predicts a 'Positive' mood, it is correct approximately 64% of the time.
  - **Recall**: With a recall of 0.83, the model is proficient at capturing most of the 'Positive' mood instances.
- **F1-Score**: The F1-scores are 0.63 for the 'Negative' class and 0.72 for the 'Positive' class, suggesting that the model is slightly better at achieving a balance between precision and recall for 'Positive' mood predictions.
- **Support**: The support values, which denote the actual number of instances for each class in the test set, are well-distributed, with 154 'Negative' and 151 'Positive' instances.
- **Macro Average**: The macro averages for precision and recall are both 0.70 and 0.69 respectively, indicating a balanced performance across the two classes.
- **Weighted Average**: The weighted averages for precision, recall, and the F1-score are all at 0.70 and 0.68, respectively. These figures consider the class distribution within the dataset, suggesting the model's performance is slightly better when class frequency is taken into account.

Overall, while the model's accuracy is relatively high, it exhibits a disparity between precision and recall, especially for the 'Negative' mood state. The model is more reliable in its 'Positive' mood predictions but shows a need for improvement in identifying 'Negative' moods, where it currently falls short. To enhance model performance, efforts could be directed at improving the detection of 'Negative' mood instances and possibly adjusting the decision threshold to balance out the recall between classes.

1.7.3 Model 3: Support Vector Machine

Generating Predictions With the optimized model from the hyperparameter tuning process, we now proceed to apply this model to predict the mood labels on the test set. This will provide us with an insight into how well the Support Vector Machine model generalizes to unseen data.

[ ]:
```
# Generate predictions for the test set using the best SVM model
y_test_pred_svc = best_svc_model.predict(X_test)
```

Computing Performance Metrics

[ ]:
```
# Compute accuracy
test_accuracy_svc = accuracy_score(y_test, y_test_pred_svc)
print(f"Test Accuracy: {test_accuracy_svc}")

# Generate a classification report
test_classification_report_svc = classification_report(y_test, y_test_pred_svc)
print("Classification Report on Test Set:\n", test_classification_report_svc)
```

```
Test Accuracy: 0.6819672131147541
Classification Report on Test Set:
               precision    recall  f1-score   support

           0       0.76      0.55      0.63       154
           1       0.64      0.82      0.72       151

    accuracy                           0.68       305
   macro avg       0.70      0.68      0.68       305
weighted avg       0.70      0.68      0.68       305
```

Interpretation of Performance Metrics The Support Vector Machine (SVM) model exhibits a test accuracy of about 68.20%. This measure indicates the proportion of total mood state predictions that the model classified correctly on the test set.

- Class 0 ('Negative' mood state):
  - **Precision**: A precision of 0.76 implies that the model's predictions are accurate approximately 76% of the time when a mood is classified as 'Negative'.
  - **Recall**: With a recall of 0.55, the model identifies 55% of the actual 'Negative' cases, suggesting a need for improvement in recognizing all 'Negative' moods.
- Class 1 ('Positive' mood state):
  - **Precision**: The precision for the 'Positive' mood predictions is 0.64, meaning that the model correctly predicts 'Positive' moods 64% of the time.

  - **Recall**: The model has a high recall of 0.82 for the ‘Positive’ class, indicating that it is adept at capturing most of the actual ‘Positive’ moods.

- **F1-Score**: The F1-score for ‘Negative’ moods stands at 0.63, while for ‘Positive’ moods, it is 0.72, showing a better balance of precision and recall for the ‘Positive’ predictions.
- **Support**: The support values represent the true number of instances for each class in the test set, with 154 cases of ‘Negative’ moods and 151 cases of ‘Positive’ moods.
- **Macro Average**: The macro average scores for precision and recall are both 0.70, indicating that the model performs equally across both classes when class distribution is not considered.
- **Weighted Average**: The weighted averages for precision, recall, and the F1-score are all 0.70 for precision and 0.68 for recall and F1-score, reflecting the model’s performance with consideration to the class distribution in the dataset.

Overall, while the SVM model’s accuracy is relatively high, there is a notable difference in performance between the classes, especially in recall for the ‘Negative’ class. The model performs well for ‘Positive’ mood predictions but could be improved for ‘Negative’ mood detection. Optimizing the model may involve investigating the features that contribute to the prediction of ‘Negative’ moods or considering alternative kernel functions or parameters to better capture the nuances of the data.

1.8 Section 8: Visualization, Discussion and Comparison of Conclusions (Second Pipeline)

The pursuit of decoding the relationship between calendar events and daily segment-based mood has yielded an intriguing confluence of data-driven insights. By delving deep into my personal calendar data spanning from June 2021 to March 21, 2024, I have constructed a predictive model to translate the quantitative measures of daily activities into qualitative mood assessments.

1.8.1 Performance Metric Comparison Table for 3 Models

Metric	BLR	XGBoost	SVM
Test Accuracy	0.6787	0.6820	0.6852
Precision (Class 0)	0.75	0.77	0.76
Recall (Class 0)	0.54	0.55	0.54
F1-Score (Class 0)	0.63	0.63	0.63
Precision (Class 1)	0.64	0.64	0.64
Recall (Class 1)	0.82	0.83	0.82
F1-Score (Class 1)	0.72	0.72	0.72

Analyzing the provided performance metrics for Binary Logistic Regression (BLR), XGBoost, and Support Vector Machine (SVM), we can draw several conclusions about their relative performances and potential reasons for these outcomes.

Test Accuracy: - SVM leads with a slight edge in accuracy over XGBoost and BLR. This could suggest that SVM’s decision boundaries, defined by the support vectors and margins, are slightly more effective at separating the classes in the test dataset.

Precision and Recall (Class 0 - 'Negative'): - XGBoost shows a marginally higher precision for Class 0, which indicates its strength in correctly identifying 'Negative' moods without as many false positives. This could be due to XGBoost's ensemble approach that builds on the errors of previous trees, potentially capturing complex patterns better. - All three models demonstrate an equal recall for Class 0. This suggests that they are all on par with each other when it comes to identifying all relevant instances of 'Negative' mood states.

Precision and Recall (Class 1 - 'Positive'): - Again, precision is uniformly distributed across all models for Class 1. This uniformity might indicate that the features relevant for predicting 'Positive' moods are well represented and equally interpreted by all three models. - XGBoost has a slightly higher recall for Class 1, albeit by a small margin. This could be due to its ability to handle unbalanced datasets inherently, given its focus on the performance of each individual tree in the ensemble.

F1-Score: - The F1-scores are identical for both classes across all models, reflecting a balanced trade-off between precision and recall. This balance is crucial in applications where both false positives and false negatives have similar costs.

Why One Model May Perform Better: - The SVM model slightly outperforms the others in accuracy, which is interesting given the equal F1-scores. This might be due to SVM's effectiveness in higher-dimensional spaces (often the case with mood prediction datasets that incorporate a large number of features from calendar data). The kernel trick allows it to find hyperplanes in a transformed feature space without the computational cost of high-dimensional mapping. - XGBoost's strength lies in its ensemble approach, which can capture complex non-linear patterns. The fact that its performance is close to SVM could indicate that the dataset has features where sequential improvements and the ensemble approach effectively capture the underlying structure. - BLR's performance being comparable to the other two models suggests that the decision boundary between 'Negative' and 'Positive' mood states can be somewhat linear, or at least can be approximated by a linear model with reasonable accuracy.

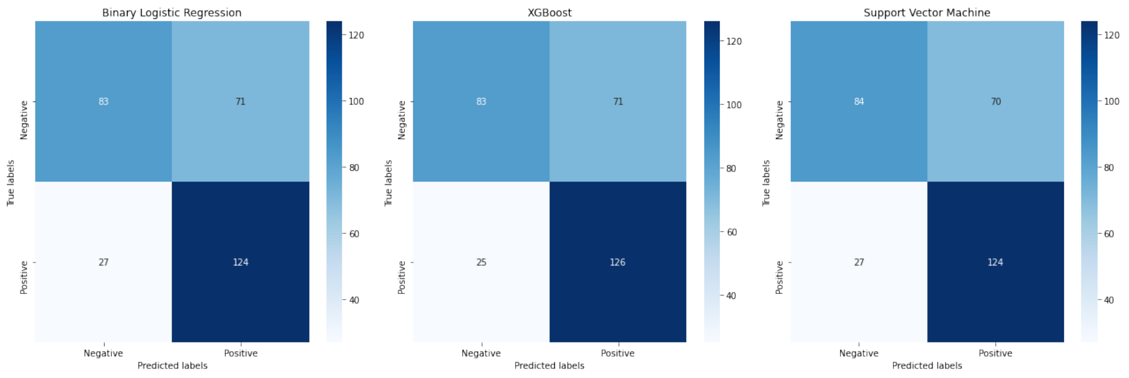
Which Model Should We Choose?:

While the differences in performance metrics are slight, they may be significant in the context of our specific application. If interpretability is key, BLR might be favored despite the marginal accuracy gain of SVM. If the highest accuracy on unseen data is the priority, and the computational cost is not an issue, SVM might be the best choice. However, if we suspect that there are complex non-linear relationships in our data that are not being captured by a linear model, and further feature engineering is not feasible, XGBoost may offer the best performance upon further tuning.

1.8.2 Confusion Matrices for 3 Models

The confusion matrices for Binary Logistic Regression, XGBoost, and Support Vector Machine showcase each model's true positive and negative rates versus their false counterparts. All three models are consistent in predicting the 'Positive' mood state, with high true positives. They also share similar challenges in false negatives for the 'Negative' mood state, mistaking quite a few 'Negative' cases as 'Positive.' The slight variations in false positives and false negatives across the models hint at differences in how they draw decision boundaries, with none significantly outperforming the others in any given aspect. This close performance reflects the earlier observed metrics, emphasizing the nuanced trade-offs between different types of errors in mood prediction. However, for better and more informed analyses, we should keep in mind that we need bigger training, validation, and test sets.

```
[ ]: from sklearn.metrics import confusion_matrix

     # Generate confusion matrices for each model

     cm_blr = confusion_matrix(y_test, y_test_pred_lr)
     cm_xgb = confusion_matrix(y_test, y_test_pred_xg)
     cm_svm = confusion_matrix(y_test, y_test_pred_svc)

     # Create a figure to hold the subplots
     fig, axes = plt.subplots(1, 3, figsize=(18, 6))

     # Titles for the models
     model_titles = ['Binary Logistic Regression', 'XGBoost', 'Support Vector␣
      ↪Machine']

     # Plot each confusion matrix
     for ax, cm, title in zip(axes, [cm_blr, cm_xgb, cm_svm], model_titles):
         sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', ax=ax)
         ax.set_title(title)
         ax.set_xlabel('Predicted labels')
         ax.set_ylabel('True labels')
         ax.set_xticklabels(['Negative', 'Positive'])
         ax.set_yticklabels(['Negative', 'Positive'])

     # Adjust layout to prevent overlap
     fig.tight_layout()

     # Display the plot
     plt.show()
```



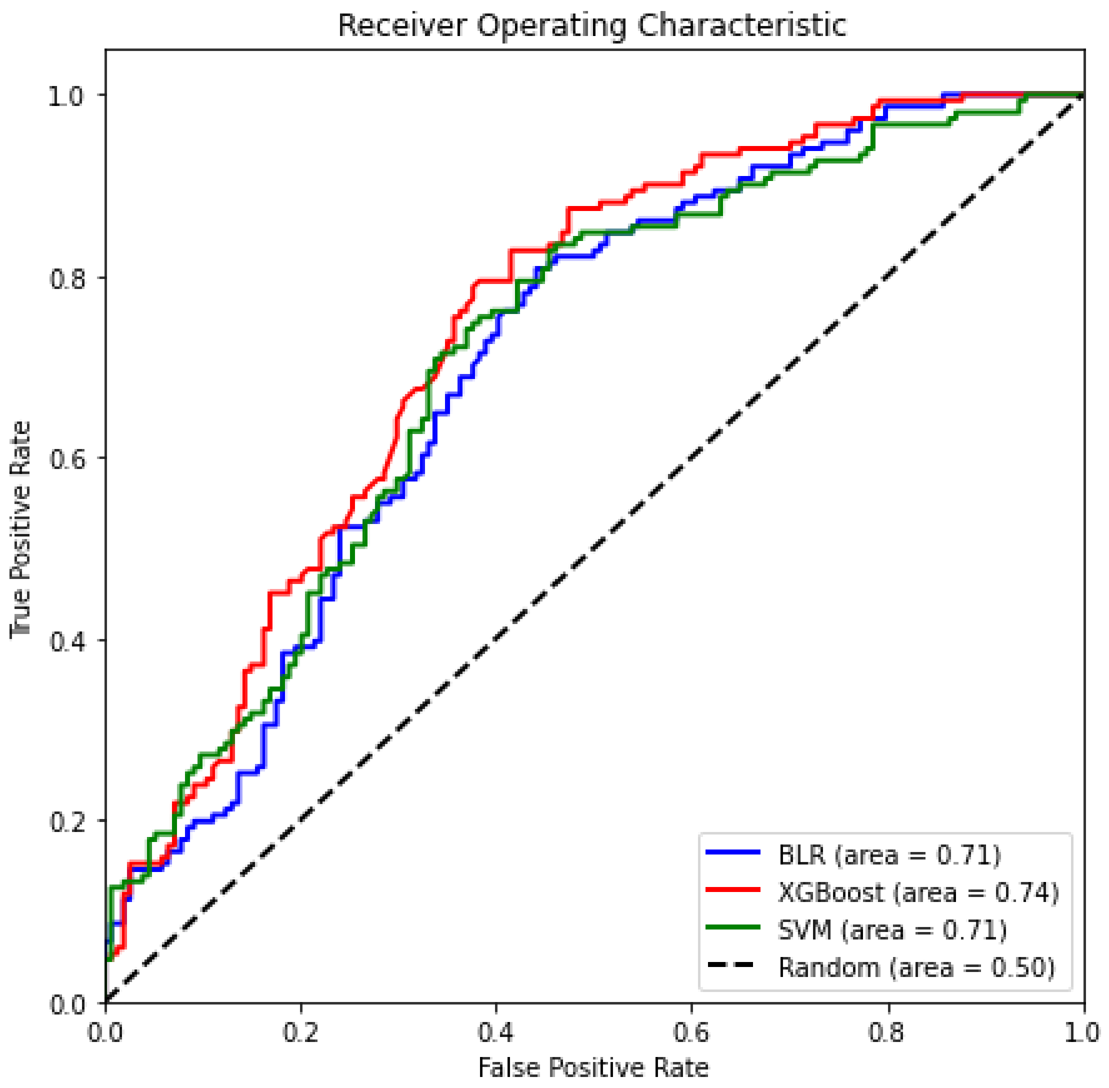
1.8.3 Comparative ROC Curves for 3 Models

The ROC curves plot the trade-off between the true positive rate and the false positive rate for the predictive models at various threshold settings. The area under the curve (AUC) is a measure of the model's ability to distinguish between the classes. Higher AUC values indicate better model performance. In this comparison:

- XGBoost (AUC = 0.74) slightly outperforms both BLR and SVM (each with AUC = 0.71), indicating it has a better balance in correctly identifying true positives while maintaining a lower rate of false positives.
- All models significantly exceed the AUC of 0.50, which represents the performance of a random classifier, showing that each model has learned meaningful patterns from the data.
- The closeness of the AUC values suggests that while XGBoost has an edge, the predictive performance of the models is relatively similar, with none vastly outperforming the others in distinguishing between the classes.

[ ]:
```
from sklearn.metrics import roc_curve, auc
from sklearn.preprocessing import label_binarize
from itertools import cycle

# Binarize the output labels for multi-class ROC curve
y_test_binarized = label_binarize(y_test, classes=[0, 1])
n_classes = y_test_binarized.shape[1]

# Assuming y_test is already binarized (i.e., it contains 0s and 1s)
models = [best_model_lr, best_xgboost_model, best_svc_model]
model_names = ['BLR', 'XGBoost', 'SVM']
colors = cycle(['blue', 'red', 'green'])

plt.figure(figsize=(7, 7))

for model, name, color in zip(models, model_names, colors):
    # Generate decision scores or probability of positive class
    if hasattr(model, "decision_function"):
        y_score = model.decision_function(X_test)
    elif hasattr(model, "predict_proba"):
        y_score = model.predict_proba(X_test)[:, 1]
    else:
        raise RuntimeError(f"The model {name} does not have decision_function␣
↪or predict_proba method.")

    # Compute ROC curve and ROC area
    fpr, tpr, _ = roc_curve(y_test, y_score)
    roc_auc = auc(fpr, tpr)

    # Plot the ROC curve
    plt.plot(fpr, tpr, color=color, lw=2, label=f'{name} (area = {roc_auc:.
↪2f})')
```

```
# Plot ROC for a random classifier
plt.plot([0, 1], [0, 1], 'k--', lw=2, label='Random (area = 0.50)')

# Customize the plot
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic')
plt.legend(loc="lower right")

plt.show()
```



1.8.4 Precision-Recall Curves for 3 Models

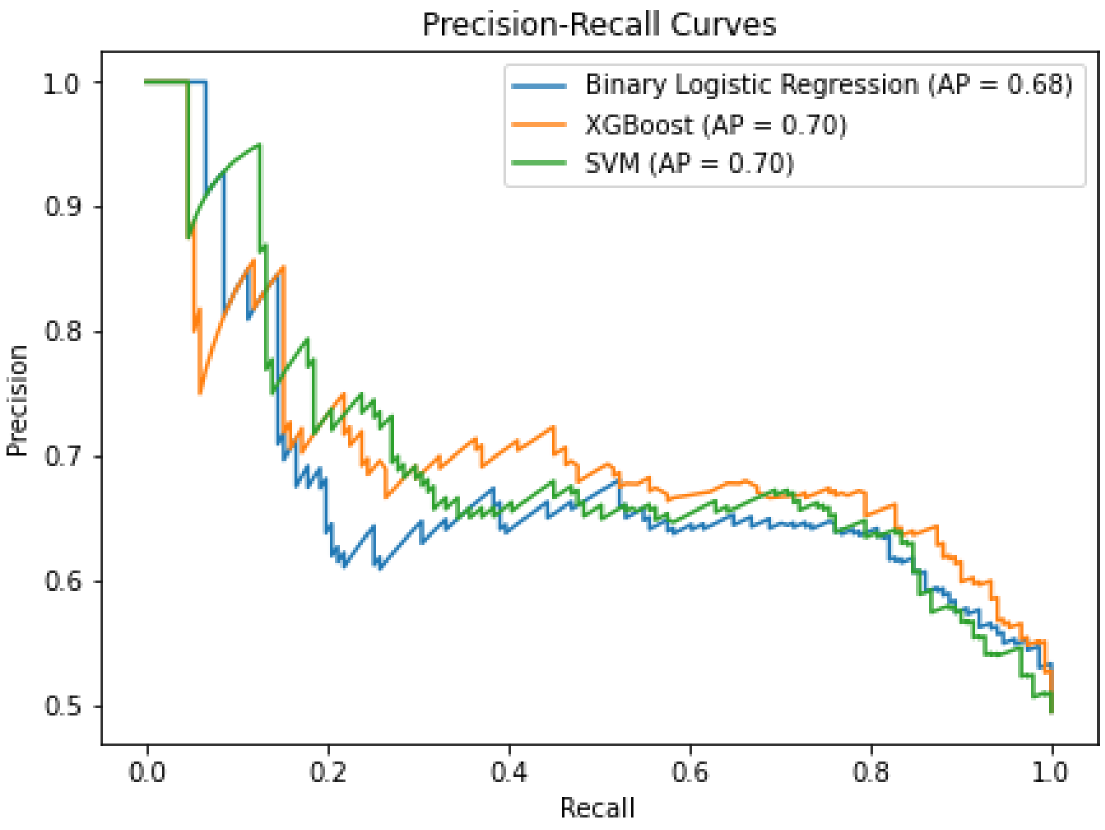
The precision-recall curves here compare the trade-offs between precision (the ability of a classifier not to label as positive a sample that is negative) and recall (the ability of a classifier to find all positive samples) across different thresholds. Each curve represents one of our models.

Binary Logistic Regression (BLR) has a lower average precision (AP) of 0.68, reflecting a generally lower performance across varying thresholds. XGBoost and Support Vector Machine (SVM) both show higher AP values of 0.70, indicating better precision and recall balance for the positive class. Both maintain higher precision at lower recall values but eventually converge with BLR at higher recall levels.

The XGBoost and SVM models' curves being higher and more to the right suggest that, for most threshold settings, they deliver a better balance of precision and recall than the BLR model. This balance is crucial in scenarios where both the reliability of the positive predictions and the model's ability to capture most positives are important.

```
[ ]: from sklearn.metrics import precision_recall_curve
     from sklearn.metrics import average_precision_score

     # For Binary Logistic Regression
     precision_lr, recall_lr, _ = precision_recall_curve(y_test, best_model_lr.
      ↪predict_proba(X_test)[:, 1])
     average_precision_lr = average_precision_score(y_test, best_model_lr.
      ↪predict_proba(X_test)[:, 1])

     # For XGBoost
     precision_xgb, recall_xgb, _ = precision_recall_curve(y_test,␣
      ↪best_xgboost_model.predict_proba(X_test)[:, 1])
     average_precision_xgb = average_precision_score(y_test, best_xgboost_model.
      ↪predict_proba(X_test)[:, 1])

     # For SVM, if it does not support probability estimates, use decision function
     # Ensure that you have scaled your features since SVM is sensitive to the␣
      ↪feature scales
     precision_svc, recall_svc, _ = precision_recall_curve(y_test, best_svc_model.
      ↪decision_function(X_test))
     average_precision_svc = average_precision_score(y_test, best_svc_model.
      ↪decision_function(X_test))

     # Now, let's plot the precision-recall curves
     plt.figure(figsize=(7, 5))

     plt.plot(recall_lr, precision_lr, label=f'Binary Logistic Regression (AP =␣
      ↪{average_precision_lr:0.2f})')
     plt.plot(recall_xgb, precision_xgb, label=f'XGBoost (AP =␣
      ↪{average_precision_xgb:0.2f})')
```

```
plt.plot(recall_svc, precision_svc, label=f'SVM (AP = {average_precision_svc:0.
 ↪2f})')

plt.xlabel('Recall')
plt.ylabel('Precision')
plt.title('Precision-Recall Curves')
plt.legend()

plt.show()
```



1.8.5 Discussing the Conclusions

In this study, we analyzed how well calendar events could predict daily moods using three different predictive models: Binary Logistic Regression (BLR), XGBoost, and Support Vector Machines (SVM). The performance was close, but SVM had a slight edge in accuracy, suggesting it's a bit better at distinguishing between moods in the given data. XGBoost also did well, especially considering it's designed to handle complex patterns, which may be present in the mood data.

The models were equally good at identifying 'Positive' moods, which might indicate that positive days have clearer signatures in the calendar data. The F1 scores across all models were consistent, indicating a balanced performance between precision and recall. This balance is crucial when both false positives and false negatives are equally important to minimize.

In summary, while no model dramatically outperformed the others, the slight differences point towards SVM as potentially the best fit for this specific task. However, BLR's simplicity could make it preferable in situations where interpretability is more critical. The choice of model would depend on the specific needs of the application, such as the trade-off between accuracy and understandability.

1.9 Section 9: Extension for the Final Pipeline - Time Series Analysis

Time series analysis has emerged as a critical tool for uncovering the dynamics within temporal data. By recognizing the inherent order and dependence of observations across time, this approach allows us to identify underlying trends and patterns and leverage them for forecasting future behavior. This analytical framework is particularly valuable for my project, where I analyze calendar data. By its very nature, my calendar data exhibits a strong sequential structure, with daily patterns that lend themselves perfectly to time series analysis techniques. Therefore, in this extension section, I will fit several machine-learning models suitable for time series classification tasks on the same data I cleaned, preprocessed, and explored before.

1.9.1 Reloading the Data

```
[ ]: from google.colab import files
     uploaded = files.upload()
```

```
<IPython.core.display.HTML object>

Saving daily_summary_chopped_labeled_enhanced.csv to
daily_summary_chopped_labeled_enhanced.csv
```

```
[ ]: # Ensure data is sequenced appropriately (chronologically) for time series␣
      ↪analysis
     import pandas as pd

     # Data is correctly prepared for time-series analysis (ordered by date and␣
      ↪segment)

     df = pd.read_csv("daily_summary_chopped_labeled_enhanced.csv")
     df.head()
```

```
[ ]:         date  segment event_ids  total_duration  avg_duration  n_events  \
     0  6/29/2021  morning         0               0           0.0         0
     1  6/29/2021  daytime         0               0           0.0         0
     2  6/29/2021  evening       325              15          15.0         1
     3  7/23/2021  morning         0               0           0.0         0
     4  7/23/2021  daytime         0               0           0.0         0

        total_attendees  avg_attendees  start_first_event  end_last_event  \
     0                0            0.0                 -1              -1
     1                0            0.0                 -1              -1
     2                2            2.0                 20              20
```

```
3                0           0.0                 -1              -1
4                0           0.0                 -1              -1

   day_of_month  day_of_week  has_event  segment_numeric      mood  \
0            29            2          0                0  Positive
1            29            2          0                1  Positive
2            29            2          1                2  Positive
3            23            5          0                0  Positive
4            23            5          0                1  Positive

                               events_descriptions  \
0                            No event descriptions.
1                            No event descriptions.
2  Engage in activities with Cecile S for collabo…
3                            No event descriptions.
4                            No event descriptions.

      events_descriptions_processed
0                               NaN
1                               NaN
2  engag activ cecil collabor effort
3                               NaN
4                               NaN
```

1.9.2 Updating the Task Description

While the core objective remains a classification task – predicting positive or negative mood based on calendar data – incorporating time series analysis injects a new dimension into our approach. This framework allows us to delve deeper, understanding and capturing the sequential dependencies inherent within the data. Traditional classification tasks often overlook these crucial temporal relationships.

Time series analysis emphasizes the temporal ordering of data points. Therefore, the selected models must inherently be able to capture time-based patterns. In this pipeline, we consider each day segment as a sequence unit. Thus, each row of data transcends its standalone nature, becoming part of a continuum where past daily segments can influence future moods. The task, therefore, evolves into a time series classification problem, where historical context plays a critical role in prediction accuracy.

In this extension, we will explore models that capitalize on these temporal relationships through the lens of time series analysis. Through my research, I identified Time Series Forests, LSTMs, and GRUs as well-suited for this enhanced task. Their inherent capacity to consider the temporal component aligns with the structure of our data.

1.9.3 Splitting the Train, Validation, and Test Sets

```
[ ]: from sklearn.feature_extraction.text import TfidfVectorizer
     from sklearn.model_selection import train_test_split
     from sklearn.preprocessing import StandardScaler
     from scipy.sparse import hstack

     # Ensure all text data is of string type
     df['events_descriptions_processed'] = df['events_descriptions_processed'].
      ↪fillna('')

     # Proceed with the vectorization as before
     vectorizer = TfidfVectorizer(max_features=1000)
     X_text_tfidf = vectorizer.fit_transform(df['events_descriptions_processed'])


     # Numeric features
     numeric_features = ['total_duration', 'avg_duration', 'n_events',␣
      ↪'total_attendees',
                         'avg_attendees', 'start_first_event', 'end_last_event',
                         'day_of_month', 'day_of_week', 'has_event',␣
      ↪'segment_numeric']

     X_numeric = df[numeric_features]

     # Standardize the numeric features
     scaler = StandardScaler()
     X_numeric_scaled = scaler.fit_transform(X_numeric)

     # Combine numeric and textual features
     X = hstack([X_numeric_scaled, X_text_tfidf])

     # Target variable
     y = df['mood'].map({'Positive': 1, 'Negative': 0}).values


     # Splitting the dataset into training (60%) and remaining (40%)
     # The random_state parameter ensures reproducibility, meaning that every time␣
      ↪the code is run,
     # the data is split the same way.
     X_train, X_remaining, y_train, y_remaining = train_test_split(
         X, y, test_size=0.4, random_state=42
     )

     # Splitting the remaining data into validation and test sets (20% each of the␣
      ↪total data)
     X_val, X_test, y_val, y_test = train_test_split(
    X_remaining, y_remaining, test_size=0.5, random_state=42
)

# Output the shapes of the resulting data splits to verify the sizes
print(f"Training features shape: {X_train.shape}")  # Expected: 60% of the data
print(f"Training labels shape: {y_train.shape}")
print(f"Validation features shape: {X_val.shape}")  # Expected: 20% of the data
print(f"Validation labels shape: {y_val.shape}")
print(f"Test features shape: {X_test.shape}")       # Expected: 20% of the data
print(f"Test labels shape: {y_test.shape}")
```

```
Training features shape: (914, 1011)
Training labels shape: (914,)
Validation features shape: (305, 1011)
Validation labels shape: (305,)
Test features shape: (305, 1011)
Test labels shape: (305,)
```

1.9.4 Extended Model Selections, Training, Predictions, and Performance Evaluations

In this section, we will be comparing various advanced time series models for our classification task. We've chosen Time Series Forest (TSF), Long Short-Term Memory networks (LSTM), and Gated Recurrent Units (GRU) for their specific attributes that cater to sequential data analysis. Each model's description, mathematical underpinnings, initiation, hyperparameter tuning, and training are methodically explained, followed by predictions and evaluations through established performance metrics. We conclude with visual demonstrations of each model's efficacy and a discussion of their analytical findings, setting the stage for a deeper understanding of time series forecasting in applied machine learning.

1.9.5 Extension Model 1 - Time Series Forest

Extension Model 1.1 - Model Selection Discussion To enhance the analysis of my dataset, I have chosen to incorporate the Time Series Forest (TSF) model. This selection is particularly well-suited for handling the inherent temporal nature of my data, which consists of chronologically ordered digital records.

TSF is an ensemble learning method derived from the Random Forest algorithm, specifically designed for time series data analysis. This model excels at capturing temporal dependencies and patterns by employing a collection of decision trees trained on various data subsets. Each tree within the forest analyzes different random segments of the time series, effectively aggregating diverse features and mitigating the risk of overfitting.

The decision to utilize TSF is driven by its robust capabilities in handling large and complex time series datasets. The model's strength lies in its interval-based approach, which extracts multiple features from random intervals across the time series. These features, encompassing statistical summaries and other derived metrics, provide a comprehensive view of the underlying temporal dynamics. This characteristic makes TSF particularly well-suited for the classification tasks within this project.

Furthermore, TSF offers flexibility by accommodating irregular sampling rates and potential missing values, which are common in real-world data. This feature is crucial for maintaining the integrity of the analysis despite data imperfections. The ensemble nature of TSF, integrating multiple decision-making pathways, enhances the model's accuracy and generalizability across the diverse temporal patterns observed in the dataset.

While TSF offers significant advantages in terms of performance and adaptability to time-series data, it is essential to acknowledge that it also introduces challenges regarding model interpretability and computational demands. Despite these considerations, implementing TSF represents a strategic choice to optimize the predictive performance for mood state classification based on temporal data patterns.

Extension Model 1.2 - Mathematical Underpinnings of the Model Interval Features

TSF generates features by summarizing statistics over various intervals of the time series. These statistics capture essential trends and variability:

- **Mean** f_1:

  The mean of the data within a specific interval t_1, t_2 is computed as:

  $$f_1(t_1, t_2) = \frac{1}{t_2 - t_1 + 1} \sum_{i=t_1}^{t_2} x_i$$

  where x_i denotes the time series value at time i, and t_1, t_2 are the interval boundaries.

- **Standard Deviation** f_2:

  This statistic measures the amount of variation or dispersion from the mean in the interval (t_1, t_2):

  $$f_2(t_1, t_2) = \sqrt{\frac{1}{t_2 - t_1} \sum_{i=t_1}^{t_2} (x_i - f_1(t_1, t_2))^2}$$

  It is defined as zero for intervals where $t_1 = t_2$ to prevent division by zero.

- **Slope of Linear Regression Line** f_3:

  The slope β_1 of the line $y = \beta_0 + \beta_1 t$ fitted to the data points in the interval t_1, t_2 is another feature:

  $$f_3(t_1, t_2) = \beta_1$$

  This line is determined using least squares regression on the points (t, x_t) for t within the interval.

Construction of Decision Trees

Each decision tree in the forest utilizes a random set of these interval-based features. The trees are constructed by recursively splitting the dataset:

Information Gain and Split Criterion

- **Entropy (H)**:

  Entropy is used to quantify the impurity or disorder within a set of items and is vital in determining effective tree splits:

$$H = -\sum_{c} p_c \log_2(p_c)$$

where p_c is the proportion of samples classified into class c. It is calculated for a node and its children to determine the best split.

- **Gain Calculation**:

  The information gain from a split is defined as the change in entropy resulting from the split:

$$\text{Gain} = H(\text{parent}) - \left(\frac{N_{\text{left}}}{N} H(\text{left}) + \frac{N_{\text{right}}}{N} H(\text{right}) \right)$$

Here, N_{left} and N_{right} are the number of samples in the left and right child nodes, respectively, and N is the total number at the current node.

- **Margin**:

  This is an auxiliary measure used to refine splits:

$$\text{Margin} = \min |x_{\text{split feature}} - \text{split point}|$$

Final Decision Criteria

The decision to split at each node in a tree uses both the information gain and the margin to maximize the model's predictive accuracy and stability:

$$\text{Split Criterion} = \text{Gain} + \alpha \times \text{Margin}$$

where α is a parameter balancing the importance of gain and margin in the decision-making process.

Extension Model 1.3 - Model Initialization, Hyperparameter Tuning, Prediction, and Performance Evaluation

[ ]:
```
pip install sktime
```

```
Collecting sktime
  Downloading sktime-0.28.0-py3-none-any.whl (21.9 MB)
                           21.9/21.9 MB
35.1 MB/s eta 0:00:00
Requirement already satisfied: numpy<1.27,>=1.21 in
/usr/local/lib/python3.10/dist-packages (from sktime) (1.25.2)
Requirement already satisfied: packaging in /usr/local/lib/python3.10/dist-
packages (from sktime) (24.0)
```

```
Requirement already satisfied: pandas<2.3.0,>=1.1 in
/usr/local/lib/python3.10/dist-packages (from sktime) (2.0.3)
Collecting scikit-base<0.8.0 (from sktime)
  Downloading scikit_base-0.7.7-py3-none-any.whl (129 kB)
                           129.9/129.9

kB 13.3 MB/s eta 0:00:00
Requirement already satisfied: scikit-learn<1.5.0,>=0.24 in
/usr/local/lib/python3.10/dist-packages (from sktime) (1.2.2)
Requirement already satisfied: scipy<2.0.0,>=1.2 in
/usr/local/lib/python3.10/dist-packages (from sktime) (1.11.4)
Requirement already satisfied: python-dateutil>=2.8.2 in
/usr/local/lib/python3.10/dist-packages (from pandas<2.3.0,>=1.1->sktime)
(2.8.2)
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-
packages (from pandas<2.3.0,>=1.1->sktime) (2023.4)
Requirement already satisfied: tzdata>=2022.1 in /usr/local/lib/python3.10/dist-
packages (from pandas<2.3.0,>=1.1->sktime) (2024.1)
Requirement already satisfied: joblib>=1.1.1 in /usr/local/lib/python3.10/dist-
packages (from scikit-learn<1.5.0,>=0.24->sktime) (1.4.0)
Requirement already satisfied: threadpoolctl>=2.0.0 in
/usr/local/lib/python3.10/dist-packages (from scikit-learn<1.5.0,>=0.24->sktime)
(3.4.0)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-
packages (from python-dateutil>=2.8.2->pandas<2.3.0,>=1.1->sktime) (1.16.0)
Installing collected packages: scikit-base, sktime
Successfully installed scikit-base-0.7.7 sktime-0.28.0
```

[ ]:
```
import numpy as np
from sktime.classification.interval_based import TimeSeriesForestClassifier

# Convert the sparse matrix to a numpy array
X_train_array = X_train.toarray()
X_test_array = X_test.toarray()

# Reshape the numpy array to have 3 dimensions (samples, time points, features)
X_train_3d = np.expand_dims(X_train_array, axis=1)
X_test_3d = np.expand_dims(X_test_array, axis=1)
```

[ ]:
```
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import accuracy_score, classification_report

# Define the parameter grid to search over
param_grid = {
    "n_estimators": [50, 100, 150],     # Number of trees in the forest
    "inner_series_length": [2, 5, 10], # Number of time series intervals
}
```

```
# Initialize the TimeSeriesForestClassifier
tsf = TimeSeriesForestClassifier(n_estimators=100, random_state=42)

# Initialize GridSearchCV with the classifier and parameter grid
grid_search = GridSearchCV(estimator=tsf, param_grid=param_grid, cv=5,␣
 ↪scoring="accuracy")

# Initialize the TimeSeriesForestClassifier
tsf = TimeSeriesForestClassifier(n_estimators=100, random_state=42)

# Fit the grid search to the training data
grid_search.fit(X_train_3d, y_train)

# Get the best parameters and best score
best_params = grid_search.best_params_
best_score = grid_search.best_score_

print("Best Parameters:", best_params)
print("Best Score:", best_score)

# Make predictions on the test data using the best estimator
best_estimator = grid_search.best_estimator_
y_pred = best_estimator.predict(X_test_3d)

# Evaluate accuracy
accuracy = accuracy_score(y_test, y_pred)
print("Time Series Forest Accuracy (after hyperparameter tuning):", accuracy)

# Generate classification report
print("Time Series Forest Classification Report (after hyperparameter tuning):")
print(classification_report(y_test, y_pred))
```

```
Best Parameters: {'inner_series_length': 10, 'n_estimators': 150}
Best Score: 0.6739626493724854
Time Series Forest Accuracy (after hyperparameter tuning): 0.6786885245901639
Time Series Forest Classification Report (after hyperparameter tuning):
              precision    recall  f1-score   support

           0       0.78      0.51      0.61       154
           1       0.63      0.85      0.72       151

    accuracy                           0.68       305
   macro avg       0.70      0.68      0.67       305
weighted avg       0.71      0.68      0.67       305
```

Interpreting the Classification Report

- **Best Parameters**: The best-performing TSF model used `inner_series_length: 10` and `n_estimators: 150`. This means that the optimal time series forest consisted of 150 trees, and each tree considered 10 time points from the series for making a decision.

- **Best Score**: The score of 0.6739 represents the highest cross-validated accuracy achieved during the hyperparameter tuning process. This value suggests that the model, with the best parameters, is able to correctly predict the class approximately 67.39% of the time on the validation set used during the cross-validation.
- **Accuracy**: After finalizing the hyperparameters and fitting the model on the full training set, the model achieved an accuracy of 0.6787 (67.87%) on the test set. This means that the model correctly predicted the outcome for roughly 68 out of every 100 samples in the test set.
- **Classification Report**: The classification report provides a breakdown of precision, recall, and F1-score for each class, as well as averages:
  - For class `0` (the negative class), the model has a precision of 0.78, which means that when it predicts the negative class, it is correct 78% of the time. However, the recall is 0.51, indicating that it only identifies 51% of all actual negative instances.
  - For class `1` (the positive class), the model has a lower precision of 0.63 but a higher recall of 0.85. This means it's less precise in its positive predictions (63% correct), but it identifies 85% of all actual positive instances.
  - F1-score for class `0` is 0.61 and for class `1` is 0.72, indicating a better harmonic mean of precision and recall for the positive class.
  - Support is number of true instances for each class in test set, which is fairly balanced (154 for class `0` and 151 for class `1`).

Extension Model 1.4 - Comparing Time Series Forest Model with Random Forest Model Since Time Series Forest model is a special form of the Random Forest model designed for time series classification tasks, let's explore the TSF's performance in comparison to RF's performance to see if accounting for potential temporal patterns in our dataset with the former model is helpful.

```
[ ]: # Implementing Random Forest Model

     from sklearn.ensemble import RandomForestClassifier
     from sklearn.model_selection import train_test_split, GridSearchCV
     from sklearn.metrics import accuracy_score, classification_report

     # Define the parameter grid for grid search
     param_grid = {
         'n_estimators': [50, 100, 150],  # Number of trees in the forest
         'max_depth': [None, 10, 20, 30],  # Maximum depth of the trees
         'min_samples_split': [2, 5, 10],  # Minimum number of samples required to␣
      ↪split a node
         'min_samples_leaf': [1, 2, 4]     # Minimum number of samples required at␣
      ↪each leaf node
     }

     # Initialize the Random Forest Classifier
```

```
rf = RandomForestClassifier(random_state=42)

# Initialize GridSearchCV
grid_search = GridSearchCV(estimator=rf, param_grid=param_grid, cv=3,␣
 ↪scoring='accuracy')

# Fit the grid search to the training data
grid_search.fit(X_train, y_train)

# Get the best parameters and best score
best_params = grid_search.best_params_
best_score = grid_search.best_score_

print("Best Parameters:", best_params)
print("Best Score:", best_score)

# Make predictions on the test data using the best model from grid search
best_rf = grid_search.best_estimator_
y_pred_rf = best_rf.predict(X_test)

# Evaluate accuracy
accuracy_rf = accuracy_score(y_test, y_pred_rf)
print("Random Forest Accuracy (after hyperparameter tuning):", accuracy_rf)

# Generate classification report
print("Random Forest Classification Report (after hyperparameter tuning):")
print(classification_report(y_test, y_pred_rf))
```

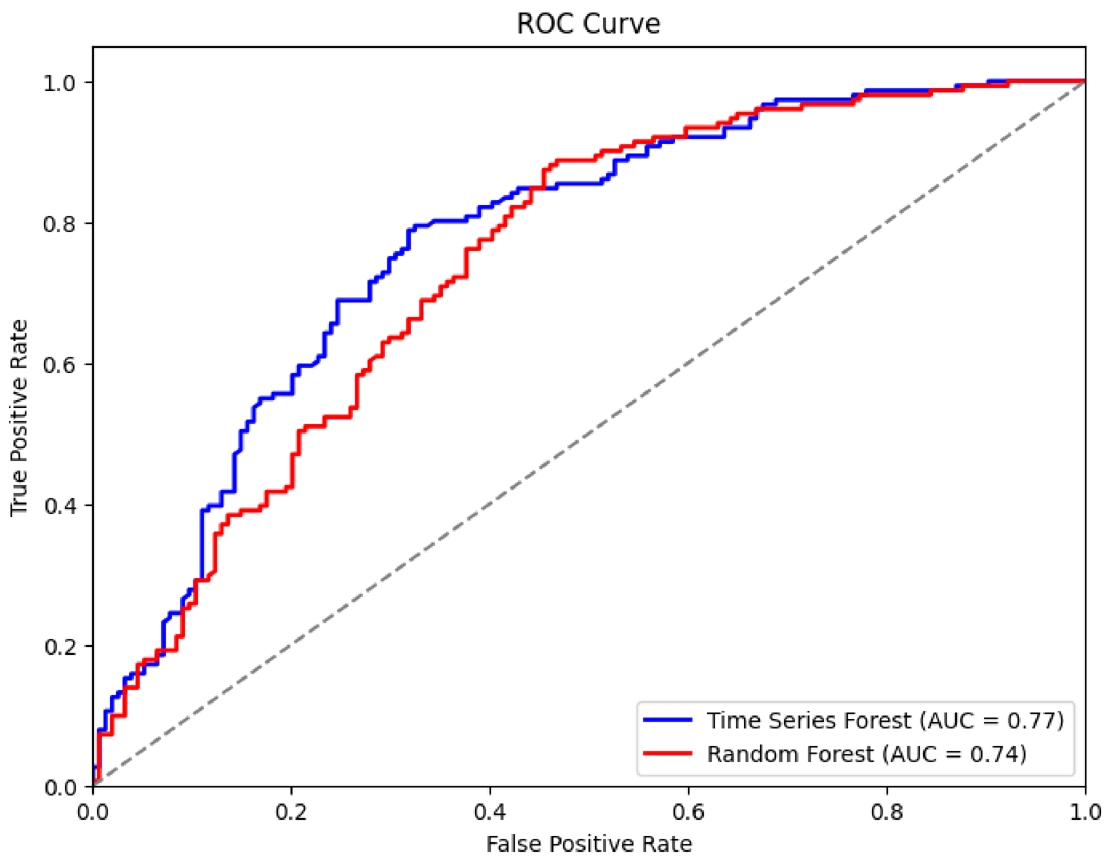
```
Best Parameters: {'max_depth': 20, 'min_samples_leaf': 2, 'min_samples_split':
10, 'n_estimators': 100}
Best Score: 0.6892795513373597
Random Forest Accuracy (after hyperparameter tuning): 0.6918032786885245
Random Forest Classification Report (after hyperparameter tuning):
              precision    recall  f1-score   support

           0       0.75      0.59      0.66       154
           1       0.66      0.79      0.72       151

    accuracy                           0.69       305
   macro avg       0.70      0.69      0.69       305
weighted avg       0.70      0.69      0.69       305
```

Interpreting the Classification Report - Best Parameters: The best-performing RF model used parameters `max_depth: 20`, `min_samples_leaf: 2`, `min_samples_split: 10`, and `n_estimators: 100`. This configuration indicates that each tree in the forest is allowed to grow to a maximum depth of 20 levels, which controls overfitting. The minimum number of samples required to be at a leaf node is 2, and a split point must have at least 10 samples.

- **Best Score**: The score of 0.6893 represents the highest accuracy score achieved during the hyperparameter tuning process through cross-validation. It suggests the model correctly predicted the outcome about 68.93% of the time on the validation data used during the cross-validation.
- **Accuracy**: After applying the best parameters and retraining the model, it achieved an accuracy of approximately 69.18% on the test dataset. This means the RF model correctly predicted the outcome for roughly 69 out of every 100 samples.
- **Classification Report**: This report gives a detailed account of performance metrics like precision, recall, F1-score, and support for each class:
  - Class `0` (negative class) has a precision of 0.75, which indicates that when the RF model predicted the negative class, it was correct 75% of the time. However, the recall is 0.59, meaning it correctly identified 59% of all actual negative instances.
  - Class `1` (positive class) has a precision of 0.66 and a recall of 0.79, indicating that the model is somewhat less precise when it predicts the positive class (66% correct) but is better at capturing the majority of positive cases (79% of actual positives).
  - F1-score, which is a balance between precision and recall, is 0.66 for class `0` and 0.72 for class `1`, indicating a better balance for the positive class.
  - Support numbers show a balanced dataset with 154 instances in class `0` and 151 in class `1`.

Now, let's compare the performance of these two models.

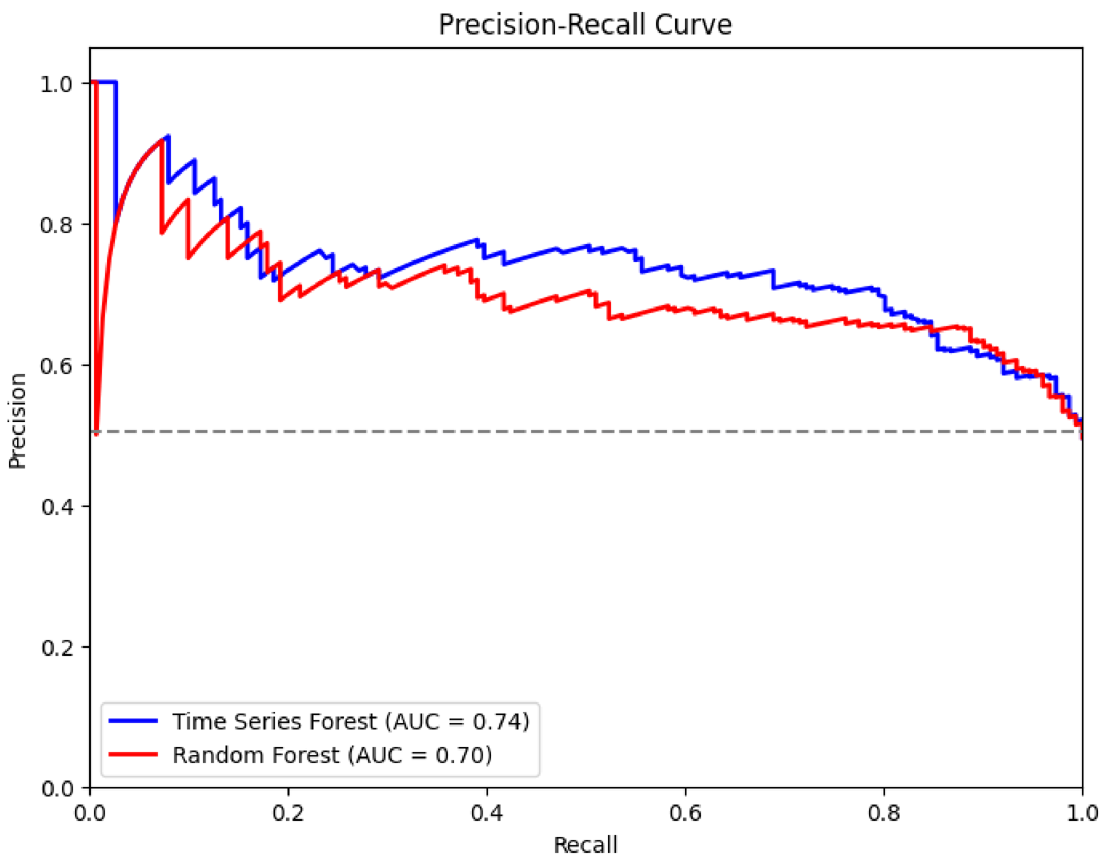
```
[ ]: from sklearn.metrics import roc_curve, roc_auc_score
     import matplotlib.pyplot as plt

     # Compute ROC curve and ROC-AUC score for Time Series Forest
     y_pred_proba_tsf = best_estimator.predict_proba(X_test_3d)[:, 1]
     fpr_tsf, tpr_tsf, _ = roc_curve(y_test, y_pred_proba_tsf)
     roc_auc_tsf = roc_auc_score(y_test, y_pred_proba_tsf)

     # Compute ROC curve and ROC-AUC score for Random Forest
     y_pred_proba_rf = best_rf.predict_proba(X_test)[:, 1]
     fpr_rf, tpr_rf, _ = roc_curve(y_test, y_pred_proba_rf)
     roc_auc_rf = roc_auc_score(y_test, y_pred_proba_rf)

     # Plot ROC curves
     plt.figure(figsize=(8, 6))
     plt.plot(fpr_tsf, tpr_tsf, color='blue', lw=2, label=f'Time Series Forest (AUC␣
      ↪= {roc_auc_tsf:.2f})')
     plt.plot(fpr_rf, tpr_rf, color='red', lw=2, label=f'Random Forest (AUC =␣
      ↪{roc_auc_rf:.2f})')
     plt.plot([0, 1], [0, 1], color='gray', linestyle='--')
     plt.xlim([0.0, 1.0])
     plt.ylim([0.0, 1.05])
     plt.xlabel('False Positive Rate')
     plt.ylabel('True Positive Rate')
plt.title('ROC Curve')
plt.legend(loc='lower right')
plt.show()
```



The area under the curve (AUC) is a measure of the model's ability to distinguish between classes. A higher AUC indicates a better performing model. In this case, the TSF, with an AUC of 0.77, suggests a superior discriminative ability compared to the Random Forest, which has an AUC of 0.74. Essentially, the TSF model is slightly better at correctly identifying positive cases without incorrectly labeling negative cases as positive across various threshold settings. The graph shows that both models perform significantly better than random guessing (indicated by the diagonal dashed line), but TSF maintains a consistent advantage over Random Forest across the full range of false positive rates.

[ ]:
```
from sklearn.metrics import precision_recall_curve, auc
import matplotlib.pyplot as plt

# Compute Precision-Recall curve and AUC for Time Series Forest
precision_tsf, recall_tsf, _ = precision_recall_curve(y_test, y_pred_proba_tsf)
auc_precision_recall_tsf = auc(recall_tsf, precision_tsf)
```

```
# Compute Precision-Recall curve and AUC for Random Forest
precision_rf, recall_rf, _ = precision_recall_curve(y_test, y_pred_proba_rf)
auc_precision_recall_rf = auc(recall_rf, precision_rf)

# Plot Precision-Recall curves
plt.figure(figsize=(8, 6))
plt.plot(recall_tsf, precision_tsf, color='blue', lw=2, label=f'Time Series␣
↪Forest (AUC = {auc_precision_recall_tsf:.2f})')
plt.plot(recall_rf, precision_rf, color='red', lw=2, label=f'Random Forest (AUC␣
↪= {auc_precision_recall_rf:.2f})')
plt.plot([0, 1], [max(y_test.mean(), 1 - y_test.mean())]*2, color='gray',␣
↪linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.title('Precision-Recall Curve')
plt.legend(loc='lower left')
plt.show()
```



The Precision-Recall Curve shows that the Time Series Forest model, with an AUC of 0.74, slightly outperforms the Random Forest, which has an AUC of 0.70. This indicates that the TSF model generally achieves a better balance between precision and recall across different thresholds. The TSF model's curve staying consistently above that of the Random Forest implies it can yield a higher precision for a given recall level, making it potentially more reliable for predicting positive classes in imbalanced datasets.

1.9.6 Extension Model 2 - Long Short-Term Memory Model

Extension Model 2.1 - Model Selection Discussion In advancing the analysis of my sequential data, I decided to incorporate Long Short-Term Memory (LSTM) networks into my project's methodology. This choice is supported by LSTM's proficiency in handling long sequence dependencies, which is pivotal given the temporal structure of our behavioral data.

LSTM is a specialized kind of Recurrent Neural Network (RNN) that is capable of learning order dependence in sequence prediction problems. This is particularly crucial in our context where the order of daily segments influences the resulting mood state predictions.

Let's talk about some of the theoretical benefits of LSTMs. Traditional RNNs face challenges related to vanishing gradients, which can lead to forgetting earlier data in long sequences. LSTM addresses this issue with its gating mechanisms, making it adept at processing data where past information significantly influences future outcomes. Additionally, the data involved in this project is inherently time-series, where each sequence's temporal characteristics might determine the mood outcome. LSTMs excel in such environments by maintaining a memory of previous data points, thus enhancing prediction accuracy. Finally, unlike simpler models that may struggle with complex sequence patterns, LSTMs can learn to recognize and even generate temporal patterns, which is beneficial for predicting mood states based on nuanced behavioral sequences.

The integration of LSTM into our data analysis pipeline brings several practical benefits as well. By effectively capturing temporal dependencies, LSTMs can improve the accuracy of predictions compared to non-sequential models. Further, LSTMs can handle variations in sequence length without the need for extensive pre-processing to standardize sequence lengths, which is a common issue in time-series analysis. Moreover, Through its recurrent connections, LSTM can use all available temporal data points, potentially turning otherwise overlooked nuances into valuable insights for mood prediction.

By selecting LSTM for our time-series analysis task, we leverage its advanced capabilities to better model the complex, time-dependent patterns present in our data, aiming for both high accuracy and robust performance in mood prediction.

Extension Model 2.2 - Mathematical Underpinnings of the Model **Core Components and Equations of LSTM:**

LSTMs manage data flow through a system of gates and cell states, where gates control information flow, and cell states store past data context.

Gates and State Equations:

1. **Forget Gate**:
    - **Purpose**: Determines the parts of the cell state to be discarded.

- **Equation**:
$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$
- **Variables**:
  - f_t: Forget gate output.
  - W_f: Weight matrix for the forget gate.
  - h_{t-1}: Output from the previous LSTM unit.
  - x_t: Input at the current timestep.
  - b_f: Bias term of the forget gate.
  - σ: Sigmoid function ensuring the gate's output is between 0 and 1.

2. **Input Gate**:
   - **Purpose**: Decides which values to update in the cell state.
   - **Equations**:
$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$
$$\tilde{C_t} = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$
   - **Variables**:
     - i_t: Output of the input gate.
     - $\tilde{C_t}$: Candidate values to add to the cell state.
     - W_i, W_C: Weights of the input gate and cell state candidate.
     - b_i, b_C: Biases of the input gate and cell state candidate.
3. **Cell State**:
   - **Purpose**: Acts as the "memory" of the LSTM, carrying relevant information throughout the processing of the sequence.
   - **Update Equation**:
$$C_t = f_t * C_{t-1} + i_t * \tilde{C_t}$$
   - **Variables**:
     - C_t: Current cell state.
     - C_{t-1}: Previous cell state.
     - $*$: Element-wise multiplication.
4. **Output Gate**:
   - **Purpose**: Determines the next hidden state reflecting the learned features.
   - **Equations**:
$$o_t = \sigma(W_o \cdot h_{t-1}, x_t + b_o)$$
$$h_t = o_t * \tanh(C_t)$$
   - **Variables**:
     - o_t: Output gate activation.
     - h_t: Current hidden state/output of the LSTM unit.
     - W_o: Weight matrix for the output gate.
     - b_o: Bias of the output gate.

Detailed Mathematical Description:

- **Sigmoid Function** (σ): Used in gate activations, it compresses values into a range between 0 and 1, effectively allowing the gate mechanisms to make "decisions" about which information to pass through.

- **Hyperbolic Tangent Function** (tanh): Provides output between -1 and 1. It is used for regulating the cell state and for output activations, helping maintain the scale of the output similar to that of the inputs.

By leveraging these components effectively, LSTMs can remember information for long periods, essential for tasks where past information is key to understanding the future. This includes complex sequence prediction tasks like language modeling and behavioral prediction based on sequential inputs.

Extension Model 2.3 - Model Initialization, Hyperparameter Tuning, Prediction, and Performance Evaluation In order to prepare our dataset for training with an LSTM model, it's crucial to reshape the data into a format that is compatible with LSTM's requirements. LSTM models expect input data in a three-dimensional format: `[samples, time_steps, features]`, where: - `samples` is the number of data points (or sequences), - `time_steps` is the sequence length of each sample, - `features` is the number of attributes used to represent each data point at each time step.

```
[ ]: def create_sequences(X, y, time_step):
         Xs, ys = [], []
         for i in range(len(X) - time_steps):
             # Reshape the features into (batch_size, time_steps, n_features)
             # Take the i to i+time_steps data for features
             Xs.append(X[i:(i + time_steps)])
             # Take the i+time_steps data for label
             ys.append(y[i + time_steps])
         return np.array(Xs), np.array(ys)

     # X is the feature matrix after converting it to dense format and y is the label
     X_dense = X.toarray()  # Convert to dense array if it fits into memory
     time_steps = 2  # Example, using previous 2 segments to predict the next

     # Create sequences for the training set
     X_train_seq, y_train_seq = create_sequences(X_train.toarray(), y_train,␣
      ↪time_steps)
     X_val_seq, y_val_seq = create_sequences(X_val.toarray(), y_val, time_steps)
     X_test_seq, y_test_seq = create_sequences(X_test.toarray(), y_test, time_steps)

     print("Training sequence shape:", X_train_seq.shape)
     print("Training label shape:", y_train_seq.shape)
     print("Validation sequence shape:", X_val_seq.shape)
     print("Validation label shape:", y_val_seq.shape)
     print("Test sequence shape:", X_test_seq.shape)
     print("Test label shape:", y_test_seq.shape)
```

```
Training sequence shape: (912, 2, 1011)
Training label shape: (912,)
Validation sequence shape: (303, 2, 1011)
Validation label shape: (303,)
```

```
Test sequence shape: (303, 2, 1011)
Test label shape: (303,)
```

To address our time series prediction task, we implemented a Long Short-Term Memory (LSTM) model using TensorFlow's Keras API. Below is a breakdown of the model architecture and its components.

```
[ ]: from tensorflow.keras.models import Sequential
     from tensorflow.keras.layers import LSTM, Dense, Dropout

     # Define an example LSTM model
     model = Sequential()
     model.add(LSTM(50, return_sequences=True, input_shape=(X_train_seq.shape[1],␣
      ↪X_train_seq.shape[2])))
     model.add(Dropout(0.2))
     model.add(LSTM(30, return_sequences=True))  # Additional LSTM layer
     model.add(Dropout(0.2))
     model.add(LSTM(20))  # Final LSTM layer does not return sequences
     model.add(Dropout(0.2))
     model.add(Dense(1, activation='sigmoid'))

     model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm (LSTM)                 (None, 2, 50)             212400

 dropout (Dropout)           (None, 2, 50)             0

 lstm_1 (LSTM)               (None, 2, 30)             9720

 dropout_1 (Dropout)         (None, 2, 30)             0

 lstm_2 (LSTM)               (None, 20)                4080

 dropout_2 (Dropout)         (None, 20)                0

 dense (Dense)               (None, 1)                 21

=================================================================
Total params: 226221 (883.68 KB)
Trainable params: 226221 (883.68 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

In the process of preparing our LSTM model for the task of mood prediction, we first compile the model using the Adam optimizer and the binary cross-entropy as the loss function. The Adam optimizer is an excellent choice for this kind of neural network because it combines the advantages of two other extensions of stochastic gradient descent, specifically Adaptive Gradient Algorithm (AdaGrad) and Root Mean Square Propagation (RMSProp). This optimizer helps resolve the vanishing learning rate problem of AdaGrad, and it is computationally efficient with little memory requirement. The binary cross-entropy loss function is used here for binary classification tasks, which calculates the cross-entropy loss between true labels and predicted labels, hence providing a measure of the model's accuracy.

To enhance the robustness of the model training, we implement an EarlyStopping callback. This is crucial to prevent overfitting, which is a common problem with models learning from high-dimensional data. The early stopping callback monitors the validation loss (val_loss), and will stop the training process if the validation loss does not improve for 20 consecutive epochs, as specified by the patience parameter. This approach helps in finding the optimal model configuration that generalizes well on unseen data. If the model begins to overfit (i.e., if it performs well on the training data but poorly on the validation data), the training will be halted to prevent further divergence.

The `model.fit()` function is then used to train the model. This function takes the training dataset as input and processes the dataset in batches of 32 samples, over 50 iterations (epochs). During each epoch, the model weights are updated to minimize the loss function, and the model's performance is evaluated on a validation set. This iterative process helps in fine-tuning the model weights to optimize performance. Importantly, by setting verbose=1, we allow the process to output detailed logs about the training progress, which helps in monitoring the model for any potential issues during training.

Through these detailed settings and configurations, the model is adeptly trained to predict mood states from sequential data, ensuring both efficiency and efficacy in its predictive performance.

```
[ ]: from tensorflow.keras import callbacks

model.compile(optimizer='adam', loss='binary_crossentropy',␣
 ↪metrics=['accuracy'])

# EarlyStopping callback to stop training when the validation loss does not␣
 ↪improve
early_stopping = callbacks.EarlyStopping(
    monitor='val_loss',     # Monitor model's validation loss
    patience=20,            # Patience is the number of epochs to wait after min␣
 ↪has been hit. After this number of no improvement, training stops.
    verbose=1,
    mode='min',             # The training will stop when the quantity monitored␣
 ↪has stopped decreasing
    restore_best_weights=True  # Restores model weights from the epoch with the␣
 ↪best value of the monitored quantity.
)

# Train the model
history = model.fit(
    X_train_seq, y_train_seq,
    epochs=50,
    batch_size=32,
    verbose=1,
    validation_data=(X_val_seq, y_val_seq),
    callbacks=[early_stopping]  # Including early stopping callback here
)
```

```
Epoch 1/50
29/29 [==============================] - 16s 107ms/step - loss: 0.6927 -
accuracy: 0.5362 - val_loss: 0.6963 - val_accuracy: 0.4356
Epoch 2/50
29/29 [==============================] - 1s 25ms/step - loss: 0.6915 - accuracy:
0.5340 - val_loss: 0.6999 - val_accuracy: 0.4356
Epoch 3/50
29/29 [==============================] - 1s 41ms/step - loss: 0.6900 - accuracy:
0.5340 - val_loss: 0.7041 - val_accuracy: 0.4356
Epoch 4/50
29/29 [==============================] - 1s 39ms/step - loss: 0.6875 - accuracy:
0.5340 - val_loss: 0.7106 - val_accuracy: 0.4356
Epoch 5/50
29/29 [==============================] - 1s 33ms/step - loss: 0.6824 - accuracy:
0.5439 - val_loss: 0.7230 - val_accuracy: 0.4389
Epoch 6/50
29/29 [==============================] - 1s 23ms/step - loss: 0.6658 - accuracy:
0.5757 - val_loss: 0.7521 - val_accuracy: 0.4389
Epoch 7/50
29/29 [==============================] - 1s 26ms/step - loss: 0.6266 - accuracy:
0.6404 - val_loss: 0.8290 - val_accuracy: 0.4356
Epoch 8/50
29/29 [==============================] - 1s 34ms/step - loss: 0.5728 - accuracy:
0.6667 - val_loss: 0.8666 - val_accuracy: 0.4653
Epoch 9/50
29/29 [==============================] - 1s 30ms/step - loss: 0.5368 - accuracy:
0.6897 - val_loss: 0.9116 - val_accuracy: 0.4587
Epoch 10/50
29/29 [==============================] - 1s 25ms/step - loss: 0.5057 - accuracy:
0.6941 - val_loss: 1.0623 - val_accuracy: 0.4323
Epoch 11/50
29/29 [==============================] - 1s 44ms/step - loss: 0.4816 - accuracy:
0.7357 - val_loss: 1.1608 - val_accuracy: 0.4455
Epoch 12/50
29/29 [==============================] - 0s 16ms/step - loss: 0.4699 - accuracy:
0.7445 - val_loss: 1.1910 - val_accuracy: 0.5017
Epoch 13/50
29/29 [==============================] - 1s 18ms/step - loss: 0.4553 - accuracy:
0.7215 - val_loss: 1.2230 - val_accuracy: 0.4752
Epoch 14/50
```

```
29/29 [==============================] - 0s 17ms/step - loss: 0.4411 - accuracy:
0.7445 - val_loss: 1.2991 - val_accuracy: 0.4653
Epoch 15/50
29/29 [==============================] - 1s 18ms/step - loss: 0.4384 - accuracy:
0.7423 - val_loss: 1.2375 - val_accuracy: 0.5116
Epoch 16/50
29/29 [==============================] - 0s 16ms/step - loss: 0.4189 - accuracy:
0.7610 - val_loss: 1.3898 - val_accuracy: 0.4686
Epoch 17/50
29/29 [==============================] - 0s 14ms/step - loss: 0.4159 - accuracy:
0.7610 - val_loss: 1.4077 - val_accuracy: 0.4983
Epoch 18/50
29/29 [==============================] - 0s 14ms/step - loss: 0.4013 - accuracy:
0.7697 - val_loss: 1.4463 - val_accuracy: 0.4719
Epoch 19/50
29/29 [==============================] - 0s 14ms/step - loss: 0.3956 - accuracy:
0.7708 - val_loss: 1.4684 - val_accuracy: 0.4884
Epoch 20/50
29/29 [==============================] - 0s 15ms/step - loss: 0.3975 - accuracy:
0.7785 - val_loss: 1.4902 - val_accuracy: 0.4917
Epoch 21/50
25/29 [========================>…] - ETA: 0s - loss: 0.3829 - accuracy:
0.7900Restoring model weights from the end of the best epoch: 1.
29/29 [==============================] - 0s 14ms/step - loss: 0.3858 - accuracy:
0.7785 - val_loss: 1.5993 - val_accuracy: 0.4818
Epoch 21: early stopping
```

The `build_lstm_model` function dynamically creates a Long Short-Term Memory (LSTM) network using Keras' Sequential API, ideal for easy layer-by-layer model construction. This function initializes an LSTM model with configurable neuron units and dropout rates to experiment with various hyperparameters. The model consists of two LSTM layers, where the first retains sequences to connect with subsequent layers, and the second compresses these sequences into a single output to prepare for the final prediction. Dropout layers are interspersed to prevent overfitting by randomly ignoring neurons during training, thus enhancing model generalizability. Finally, a dense layer with a sigmoid activation function determines the binary outcome. This setup, compiled with the Adam optimizer and binary cross-entropy loss, allows for robust experimentation across different LSTM configurations to optimize accuracy in binary classification tasks.

[ ]:
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Dropout

def build_lstm_model(units, dropout_rate):
    model = Sequential([
        LSTM(units, return_sequences=True, input_shape=(X_train_seq.shape[1],␣
 ↪X_train_seq.shape[2])),
        Dropout(dropout_rate),
        LSTM(units, return_sequences=False),
        Dropout(dropout_rate),
        Dense(1, activation='sigmoid')
    ])
    model.compile(optimizer='adam', loss='binary_crossentropy',␣
 ↪metrics=['accuracy'])
    return model
```

Next up, hyperparameter tuning is a crucial step in optimizing a deep learning model. The process involves experimenting with different combinations of hyperparameters to find the most effective configuration for the model. Below is an explanation of the hyperparameter tuning implemented for the LSTM model using a grid search approach.

In this code snippet, we define a function `build_lstm_model` that initializes a Sequential LSTM model with specified units and dropout rates. This function simplifies the process of model instantiation, allowing us to easily modify hyperparameters during our tuning process.

We define a param_grid dictionary containing arrays of values for different hyperparameters we wish to tune: LSTM units, dropout rates, and the number of time steps. These parameters affect the model's capacity, its ability to generalize, and its temporal sensitivity, respectively.

Later, we conduct a grid search to explore different combinations of the defined hyperparameters. For each combination, we:

- Generate training, validation, and testing sequences with the current time_steps.
- Build and compile an LSTM model.
- Fit the model on the training data while validating it on the validation set and apply early stopping to prevent overfitting.
- Record the model's performance and update our records if the current model outperforms previous ones.

By the end of this grid search, we identify the best performing model configuration (best_overall_config), which provides us insights into the most effective hyperparameters for our specific dataset and task. This systematic approach to hyperparameter tuning is essential for optimizing model performance and ensuring robustness in predictions.

```
[ ]: import numpy as np
     import pandas as pd
     import matplotlib.pyplot as plt
     from sklearn.model_selection import train_test_split
     from sklearn.preprocessing import StandardScaler
     from sklearn.metrics import accuracy_score, precision_score, recall_score,␣
      ↪f1_score, roc_auc_score
     from scipy.sparse import hstack
     from tensorflow.keras.models import Sequential
     from tensorflow.keras.layers import LSTM, Dense, Dropout
     from tensorflow.keras import callbacks  # Import callbacks module

     # Define the parameter grid
     param_grid = {
         'units': [50, 100, 150],
         'dropout_rate': [0.1, 0.2, 0.3],
    'time_steps': [1, 2, 3]  # Including time_steps as a hyperparameter
}

best_overall_config = None
best_overall_val_accuracy = 0
history_records = []

for time_steps in param_grid['time_steps']:
    X_train_seq, y_train_seq = create_sequences(X_train.toarray(), y_train,␣
 ↪time_steps)
    X_val_seq, y_val_seq = create_sequences(X_val.toarray(), y_val, time_steps)
    X_test_seq, y_test_seq = create_sequences(X_test.toarray(), y_test,␣
 ↪time_steps)

    for units in param_grid['units']:
        for dropout in param_grid['dropout_rate']:
            print(f"Testing {units} units and {dropout} dropout with␣
 ↪{time_steps} time steps.")

            model = build_lstm_model(units=units, dropout_rate=dropout)

            # Define a list of callbacks
            callback_list = [callbacks.EarlyStopping(monitor='val_loss',␣
 ↪patience=5, restore_best_weights=True)]

            history = model.fit(X_train_seq, y_train_seq,␣
 ↪validation_data=(X_val_seq, y_val_seq),
                                epochs=30, batch_size=32,␣
 ↪callbacks=callback_list, verbose=0)

            val_accuracy = max(history.history['val_accuracy'])
            history_records.append((time_steps, units, dropout, history.
 ↪history))

            if val_accuracy > best_overall_val_accuracy:
                best_overall_val_accuracy = val_accuracy
                best_overall_config = (time_steps, units, dropout)
                best_model = model

print(f"Best overall config: {best_overall_config} with validation accuracy␣
 ↪{best_overall_val_accuracy}")
```

```
Testing 50 units and 0.1 dropout with 1 time steps.
Testing 50 units and 0.2 dropout with 1 time steps.
Testing 50 units and 0.3 dropout with 1 time steps.
Testing 100 units and 0.1 dropout with 1 time steps.
Testing 100 units and 0.2 dropout with 1 time steps.
     Testing 100 units and 0.3 dropout with 1 time steps.
     Testing 150 units and 0.1 dropout with 1 time steps.
     Testing 150 units and 0.2 dropout with 1 time steps.
     Testing 150 units and 0.3 dropout with 1 time steps.
     Testing 50 units and 0.1 dropout with 2 time steps.
     Testing 50 units and 0.2 dropout with 2 time steps.
     Testing 50 units and 0.3 dropout with 2 time steps.
     Testing 100 units and 0.1 dropout with 2 time steps.
     Testing 100 units and 0.2 dropout with 2 time steps.
     Testing 100 units and 0.3 dropout with 2 time steps.
     Testing 150 units and 0.1 dropout with 2 time steps.
     Testing 150 units and 0.2 dropout with 2 time steps.
     Testing 150 units and 0.3 dropout with 2 time steps.
     Testing 50 units and 0.1 dropout with 3 time steps.
     Testing 50 units and 0.2 dropout with 3 time steps.
     Testing 50 units and 0.3 dropout with 3 time steps.
     Testing 100 units and 0.1 dropout with 3 time steps.
     Testing 100 units and 0.2 dropout with 3 time steps.
     Testing 100 units and 0.3 dropout with 3 time steps.
     Testing 150 units and 0.1 dropout with 3 time steps.
     Testing 150 units and 0.2 dropout with 3 time steps.
     Testing 150 units and 0.3 dropout with 3 time steps.
     Best overall config: (3, 100, 0.2) with validation accuracy 0.5397350788116455
```

After identifying the best hyperparameters from our tuning process, the next step is to build and train the LSTM model using these optimal settings. This process ensures that we are using the most effective configuration discovered during hyperparameter tuning.

```
[ ]: best_time_steps = best_overall_config[1]
X_train_seq_lstm, y_train_seq_lstm = create_sequences(X_train.toarray(),␣
 ↪y_train, best_time_steps)
X_val_seq_lstm, y_val_seq_lstm = create_sequences(X_val.toarray(), y_val,␣
 ↪best_time_steps)
X_test_seq_lstm, y_test_seq_lstm = create_sequences(X_test.toarray(), y_test,␣
 ↪best_time_steps)

# Rebuilding the model with the best parameters
model = build_lstm_model(units = best_overall_config[1], dropout_rate =␣
 ↪best_overall_config[2])

# Train the model
history_lstm = model.fit(X_train_seq_lstm, y_train_seq_lstm, epochs=30,␣
 ↪batch_size=32, validation_data=(X_val_seq_lstm, y_val_seq_lstm))
```

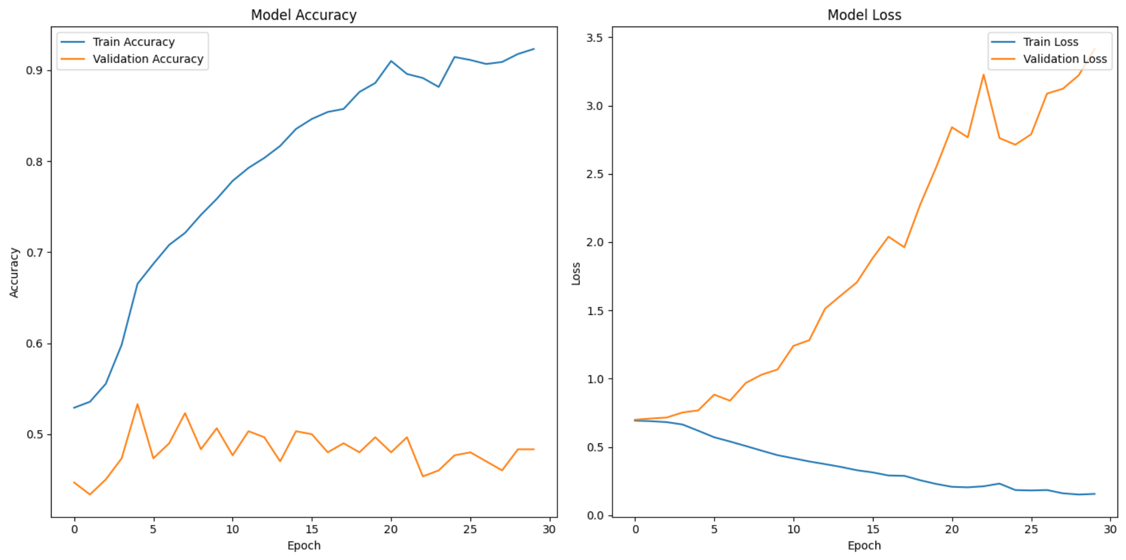
```
Epoch 1/30
29/29 [==============================] - 12s 134ms/step - loss: 0.6924 -
accuracy: 0.5291 - val_loss: 0.6987 - val_accuracy: 0.4470
Epoch 2/30
```

```
29/29 [==============================] - 1s 41ms/step - loss: 0.6884 - accuracy:
0.5357 - val_loss: 0.7076 - val_accuracy: 0.4338
Epoch 3/30
29/29 [==============================] - 1s 41ms/step - loss: 0.6814 - accuracy:
0.5554 - val_loss: 0.7148 - val_accuracy: 0.4503
Epoch 4/30
29/29 [==============================] - 1s 47ms/step - loss: 0.6641 - accuracy:
0.5982 - val_loss: 0.7517 - val_accuracy: 0.4735
Epoch 5/30
29/29 [==============================] - 1s 35ms/step - loss: 0.6181 - accuracy:
0.6652 - val_loss: 0.7675 - val_accuracy: 0.5331
Epoch 6/30
29/29 [==============================] - 1s 31ms/step - loss: 0.5706 - accuracy:
0.6872 - val_loss: 0.8827 - val_accuracy: 0.4735
Epoch 7/30
29/29 [==============================] - 1s 21ms/step - loss: 0.5394 - accuracy:
0.7080 - val_loss: 0.8382 - val_accuracy: 0.4901
Epoch 8/30
29/29 [==============================] - 1s 21ms/step - loss: 0.5062 - accuracy:
0.7212 - val_loss: 0.9684 - val_accuracy: 0.5232
Epoch 9/30
29/29 [==============================] - 1s 26ms/step - loss: 0.4723 - accuracy:
0.7409 - val_loss: 1.0290 - val_accuracy: 0.4834
Epoch 10/30
29/29 [==============================] - 1s 22ms/step - loss: 0.4393 - accuracy:
0.7585 - val_loss: 1.0668 - val_accuracy: 0.5066
Epoch 11/30
29/29 [==============================] - 1s 21ms/step - loss: 0.4163 - accuracy:
0.7783 - val_loss: 1.2395 - val_accuracy: 0.4768
Epoch 12/30
29/29 [==============================] - 1s 21ms/step - loss: 0.3933 - accuracy:
0.7925 - val_loss: 1.2813 - val_accuracy: 0.5033
Epoch 13/30
29/29 [==============================] - 1s 21ms/step - loss: 0.3738 - accuracy:
0.8035 - val_loss: 1.5124 - val_accuracy: 0.4967
Epoch 14/30
29/29 [==============================] - 1s 21ms/step - loss: 0.3532 - accuracy:
0.8167 - val_loss: 1.6095 - val_accuracy: 0.4702
Epoch 15/30
29/29 [==============================] - 1s 22ms/step - loss: 0.3291 - accuracy:
0.8353 - val_loss: 1.7046 - val_accuracy: 0.5033
Epoch 16/30
29/29 [==============================] - 1s 22ms/step - loss: 0.3131 - accuracy:
0.8463 - val_loss: 1.8822 - val_accuracy: 0.5000
Epoch 17/30
29/29 [==============================] - 1s 21ms/step - loss: 0.2905 - accuracy:
0.8540 - val_loss: 2.0394 - val_accuracy: 0.4801
Epoch 18/30
```

```
29/29 [==============================] - 1s 21ms/step - loss: 0.2878 - accuracy:
0.8573 - val_loss: 1.9619 - val_accuracy: 0.4901
Epoch 19/30
29/29 [==============================] - 1s 24ms/step - loss: 0.2556 - accuracy:
0.8760 - val_loss: 2.2743 - val_accuracy: 0.4801
Epoch 20/30
29/29 [==============================] - 1s 21ms/step - loss: 0.2288 - accuracy:
0.8858 - val_loss: 2.5463 - val_accuracy: 0.4967
Epoch 21/30
29/29 [==============================] - 1s 21ms/step - loss: 0.2076 - accuracy:
0.9100 - val_loss: 2.8410 - val_accuracy: 0.4801
Epoch 22/30
29/29 [==============================] - 1s 30ms/step - loss: 0.2038 - accuracy:
0.8957 - val_loss: 2.7675 - val_accuracy: 0.4967
Epoch 23/30
29/29 [==============================] - 1s 36ms/step - loss: 0.2114 - accuracy:
0.8913 - val_loss: 3.2271 - val_accuracy: 0.4536
Epoch 24/30
29/29 [==============================] - 1s 34ms/step - loss: 0.2312 - accuracy:
0.8814 - val_loss: 2.7620 - val_accuracy: 0.4603
Epoch 25/30
29/29 [==============================] - 1s 22ms/step - loss: 0.1838 - accuracy:
0.9144 - val_loss: 2.7131 - val_accuracy: 0.4768
Epoch 26/30
29/29 [==============================] - 1s 21ms/step - loss: 0.1811 - accuracy:
0.9111 - val_loss: 2.7896 - val_accuracy: 0.4801
Epoch 27/30
29/29 [==============================] - 1s 25ms/step - loss: 0.1839 - accuracy:
0.9067 - val_loss: 3.0876 - val_accuracy: 0.4702
Epoch 28/30
29/29 [==============================] - 1s 22ms/step - loss: 0.1600 - accuracy:
0.9089 - val_loss: 3.1231 - val_accuracy: 0.4603
Epoch 29/30
29/29 [==============================] - 1s 21ms/step - loss: 0.1512 - accuracy:
0.9177 - val_loss: 3.2216 - val_accuracy: 0.4834
Epoch 30/30
29/29 [==============================] - 1s 24ms/step - loss: 0.1557 - accuracy:
0.9232 - val_loss: 3.4142 - val_accuracy: 0.4834
```

[ ]:
```
import matplotlib.pyplot as plt
import seaborn as sns

def plot_history(history):
    '''Function that plots the history of LSTM model over multiple epochs.'''
    plt.figure(figsize=(14, 7))

    # Plotting Accuracy
    plt.subplot(1, 2, 1)
    plt.plot(history.history['accuracy'], label='Train Accuracy')
    plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
    plt.title('Model Accuracy')
    plt.ylabel('Accuracy')
    plt.xlabel('Epoch')
    plt.legend(loc='upper left')

    # Plotting Loss
    plt.subplot(1, 2, 2)
    plt.plot(history.history['loss'], label='Train Loss')
    plt.plot(history.history['val_loss'], label='Validation Loss')
    plt.title('Model Loss')
    plt.ylabel('Loss')
    plt.xlabel('Epoch')
    plt.legend(loc='upper right')

    plt.tight_layout()  # Adjust layout to make it more readable
    plt.show()
```

[ ]:
```
plot_history(history_lstm)
```



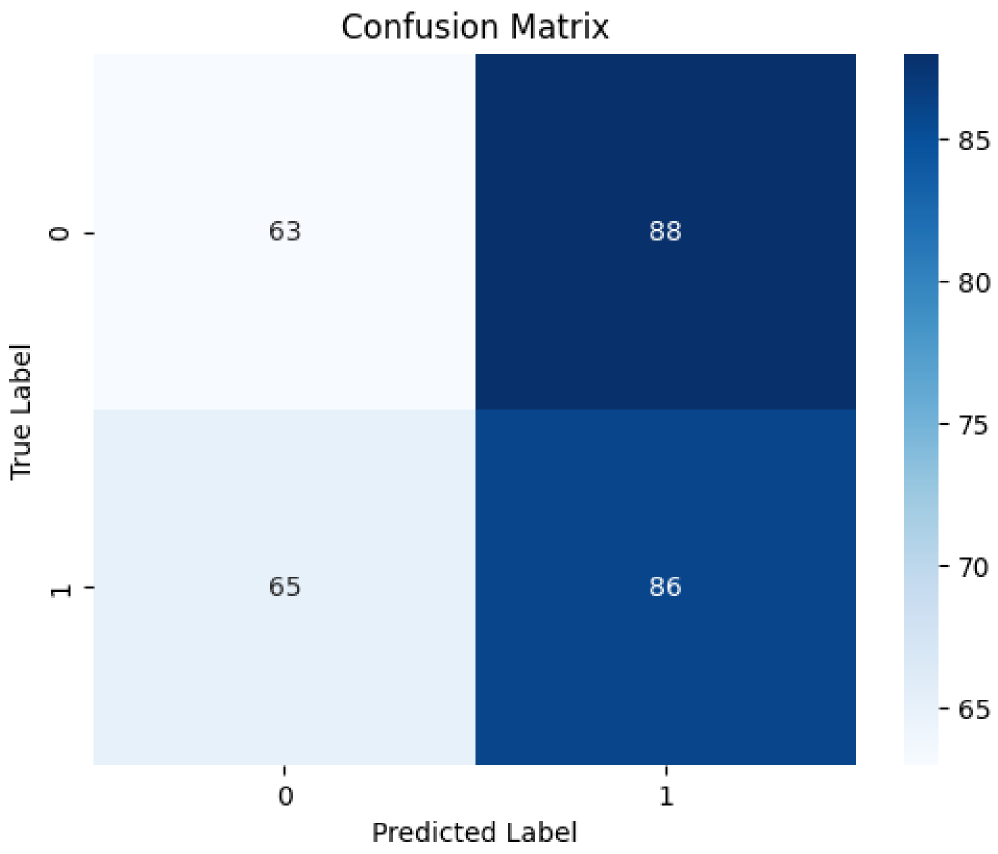
Now, let's make a prediction on our test set and evaluate the performance of our LSTM implementation through visualizations.

[ ]:
```
# Make predictions
y_pred_probs_lstm = model.predict(X_test_seq_lstm)
y_preds_lstm = (y_pred_probs_lstm > 0.5).astype(int)  # Convert probabilities␣
  ↪to binary output
```

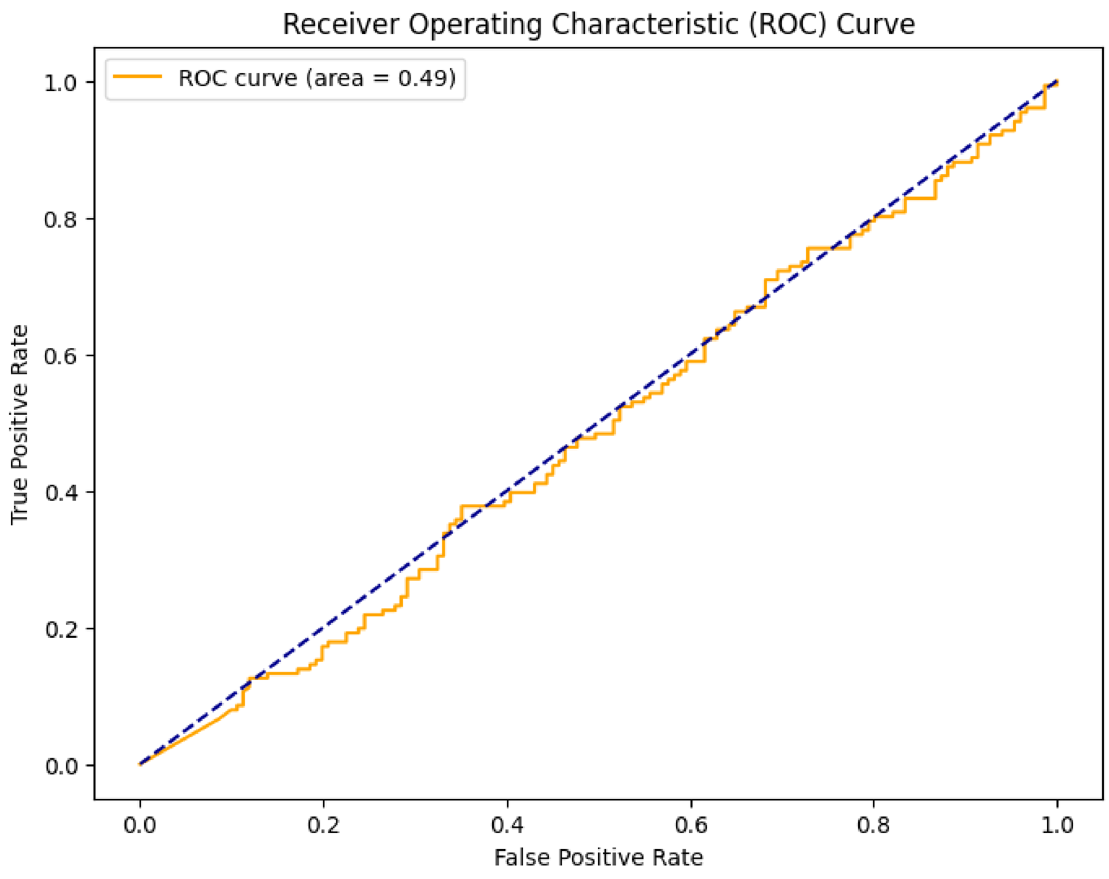
```
10/10 [==============================] - 2s 18ms/step
```

[ ]:
```
from sklearn.metrics import confusion_matrix

# Classification report
report = classification_report(y_test_seq_lstm, y_preds_lstm)
print(report)

# Confusion Matrix
cm = confusion_matrix(y_test_seq_lstm, y_preds_lstm)
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.title('Confusion Matrix')
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.show()
```

```
              precision    recall  f1-score   support

           0       0.49      0.42      0.45       151
           1       0.49      0.57      0.53       151

    accuracy                           0.49       302
   macro avg       0.49      0.49      0.49       302
weighted avg       0.49      0.49      0.49       302
```



```
[ ]: from sklearn.metrics import roc_curve, roc_auc_score

     # Plot ROC Curve
     fpr, tpr, thresholds = roc_curve(y_test_seq_lstm, y_pred_probs_lstm)

     # Calculate the AUC (Area under the ROC Curve)
     roc_auc_lstm = roc_auc_score(y_test_seq_lstm, y_pred_probs_lstm)

     plt.figure(figsize=(8, 6))
     plt.plot(fpr, tpr, color='orange', label=f'ROC curve (area = {roc_auc_lstm:.
      ↪2f})')
     plt.plot([0, 1], [0, 1], color='darkblue', linestyle='--')
     plt.xlabel('False Positive Rate')
     plt.ylabel('True Positive Rate')
     plt.title('Receiver Operating Characteristic (ROC) Curve')
     plt.legend()
     plt.show()
```



Discussion of Performance Results for LSTM

The LSTM model's performance, based on the metrics provided, seems to indicate overfitting as a potential issue. The training accuracy reaches quite high levels, near 90%, while the validation accuracy fluctuates around 50%, which is only marginally better than random guessing. This discrepancy suggests that the model is learning patterns specific to the training data that do not generalize well to unseen data.

The model loss graphs reinforce this interpretation; the training loss steadily decreases, indicating that the model is becoming more confident in its training predictions. However, the validation loss is erratic and generally trends upwards, which is a classic sign of overfitting.

The confusion matrix and the Receiver Operating Characteristic (ROC) curve provide further insights. The confusion matrix shows that the model does not have a significant bias toward predicting one class over the other, which is good. However, the precision, recall, and F1-score for both classes are around 0.5, indicating that the model is not effective at distinguishing between the classes.

The ROC curve further confirms the model's lackluster performance with an area under the curve (AUC) of approximately 0.49, which is close to the AUC of 0.5 that would be expected from random chance. This implies that the model does not have good separability; it does not distinguish between the positive and negative classes well.

In summary, while the LSTM model appears to learn from the training data, it fails to generalize this learning to the validation set, resulting in poor predictive performance on the data it has not seen before. This issue could potentially be addressed by collecting more data, applying more rigorous regularization techniques, introducing dropout layers, or experimenting with simpler model architectures that are less prone to overfitting.

1.9.7 Extension Model 3 - Gated Recurrent Unit

Extension Model 3.1 - Model Selection Discussion In selecting the appropriate model for the time-series prediction tasks involved in this project, the Gated Recurrent Unit (GRU) offers a balance between complexity and performance. GRUs are an advancement over traditional Recurrent Neural Networks (RNNs) and solve some of the critical challenges faced by its predecessor, such as the vanishing gradient problem.

GRUs are designed to adaptively capture dependencies of different time scales. They achieve this through gating mechanisms which regulate the flow of information. These gates are capable of learning which data in a sequence is important to keep or discard, thereby preserving long-term dependencies without the risk of vanishing gradients. This makes GRUs particularly fitting for our dataset, where understanding the long-term context is crucial for predicting mood states from temporal event data.

The GRU modifies the typical RNN architecture by combining the forget and input gates into a single "update gate," and also merging the cell state and hidden state, thus simplifying the model and reducing the computational burden. This is particularly beneficial for our use case as it allows for faster training times without compromising the ability to model complex patterns in time-series data.

Furthermore, GRUs have shown a robust performance on a variety of sequence modeling problems, often outperforming complex models like LSTMs on tasks where data points are closely related, as is the case with time-stamped mood prediction. Given their simpler structure and the reduced risk of overfitting, GRUs present a potent model choice for our project's requirements to efficiently process and predict based on structured time-series data.

The combination of flexibility, efficiency, and the capability to model long dependencies makes GRUs an excellent candidate for further exploring the nuanced relationships in our dataset. This aligns with my project's goal to achieve high accuracy in mood state prediction while maintaining manageable computational costs.

Extension Model 3.2 - Mathematical Underpinnings of the Model Core Equations of GRU

The GRU modifies the standard recurrent neural network architecture by introducing two gates: a reset gate and an update gate. These gates determine how information is transferred to future steps, which helps in preserving the error that can be backpropagated through time and layers.

1. **Update Gate**:
$$z_t = \sigma(W_z \cdot [h_{t-1}, x_t] + b_z)$$
   Here, z_t is the update gate's activation vector. W_z is the weight matrix for the update gate, h_{t-1} is the previous hidden state, x_t is the input at the current step, and b_z is the bias.

2. **Reset Gate**:
$$r_t = \sigma(W_r \cdot [h_{t-1}, x_t] + b_r)$$
r_t is the reset gate's activation vector which determines how much of the past information to forget. W_r is the weight matrix for the reset gate, and b_r is the bias.

3. **Candidate Hidden State**:
$$\tilde{h}_t = \tanh(W \cdot [r_t * h_{t-1}, x_t] + b)$$
The candidate hidden state, $\tilde{h}_t$, is computed using a tanh activation function which combines the input and the past hidden state, gated by the reset state. Here, $*$ denotes element-wise multiplication.

4. **Final Hidden State**:
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$
The actual hidden state for the current timestep is then calculated using the update gate. The update gate controls the extent to which the new state is just the old state h_{t-1} and how much to add the new candidate hidden state $\tilde{h}_t$.

Detailed Variable Explanation

- x_t: Input vector at time step t.
- h_{t-1}: Hidden state vector from the previous time step $t-1$.
- z_t: Update gate vector at time step t, controlling how much of the past information from previous time steps is kept.
- r_t: Reset gate vector at time step t, deciding how much of the past information to forget.
- $\tilde{h}_t$: Candidate hidden state vector at time step t, blending the input and the past hidden state.
- h_t: New hidden state vector at time step t.

Operational Insights

- The **update gate** z_t helps the model to determine the amount of the past information (from previous timesteps) that needs to be passed along to the future.
- The **reset gate** r_t can be seen as a way to make the model forget the previously computed state information, which is deemed irrelevant for the current state calculation.

By effectively using gates to control the flow of information, GRU can model temporal sequences and their long-range dependencies more accurately than standard RNNs, which is crucial for tasks like time-series analysis, natural language processing, and more dynamic systems modeling. This gating mechanism, inherent to GRUs, allows it to avoid the vanishing gradient problem, providing a way to preserve information over longer periods without degradation.

Extension Model 3.3 - Model Initialization, Hyperparameter Tuning, Prediction, and Performance Evaluation The implementation below outlines the setup of a Gated Recurrent Unit (GRU) model using the TensorFlow/Keras framework, which is particularly suited for handling sequences and time-series data. The model is designed as follows:

- **Configuration**: It includes GRU layers configured with 64 units to process sequences, with a dropout rate of 20% after each GRU layer to prevent overfitting by randomly omitting features during training.

- **Architecture**: The model starts with a GRU layer that returns sequences to the next layer, making it suitable for stacking multiple recurrent layers. It is followed by another GRU layer that simplifies the output to the last timestep only, useful for making final predictions. This setup helps in learning from the temporal dependencies in the data, crucial for accurate forecasting in time-series analysis.
- **Output**: The final layer is a densely-connected neural layer that uses a sigmoid activation function to output a probability, indicating the likelihood of belonging to one of two classes (binary classification).

```
[ ]: from tensorflow.keras.models import Sequential
     from tensorflow.keras.layers import GRU, Dense, Dropout

     # Model configuration
     gru_units = 64  # Number of units in the GRU layer
     dropout_rate = 0.2  # Dropout rate for regularization

     # Build an example GRU model
     model = Sequential()
     model.add(GRU(units=gru_units, return_sequences=True, input_shape=(X_train_seq.
       ↪shape[1], X_train_seq.shape[2])))
     model.add(Dropout(dropout_rate))
     model.add(GRU(units=gru_units, return_sequences=False))  # Stack another GRU␣
       ↪layer
     model.add(Dropout(dropout_rate))
     model.add(Dense(1, activation='sigmoid'))  # Output layer with sigmoid␣
       ↪activation for binary classification
     model.summary()
```

```
Model: "sequential_29"

_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 gru (GRU)                   (None, 3, 64)             206784

 dropout_59 (Dropout)        (None, 3, 64)             0

 gru_1 (GRU)                 (None, 64)                24960

 dropout_60 (Dropout)        (None, 64)                0

 dense_29 (Dense)            (None, 1)                 65

=================================================================
Total params: 231809 (905.50 KB)
Trainable params: 231809 (905.50 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

In this part of the implementation, we first compile the GRU model using the Adam optimizer. Adam is widely used due to its adaptive learning rate capabilities, which is especially beneficial in training neural networks where different parameters may require different learning adjustments. The binary_crossentropy loss function is chosen, fitting for binary classification tasks, and we track the accuracy during the training and validation phases.

Next, we set up two crucial callbacks—ModelCheckpoint and EarlyStopping. The ModelCheckpoint callback is configured to save the model weights only when there is an improvement in validation accuracy (val_accuracy), ensuring that the best model is stored without manually monitoring each epoch. This automation aids significantly in managing long training sessions.

The EarlyStopping callback is designed to halt the training process if the validation loss (val_loss) does not decrease for 20 consecutive epochs. By monitoring the validation loss and restoring weights from the best state, this callback prevents the model from overfitting to the training data, which is a common issue in complex neural networks. This setup not only helps in maintaining an optimal model state but also reduces computational wastage by stopping early if no further improvements are seen.

The training process is then executed using the specified configurations. The model trains on batches of 32 samples (batch_size=32) for a maximum of 50 epochs, although it can stop earlier if the EarlyStopping criteria are met. During training, progress updates are printed out thanks to verbose=1, providing insights into the model's learning at each epoch. Using both training and validation datasets helps in validating the model's performance on unseen data continuously, which is critical for assessing the generalizability of the trained model.

```
[ ]: from tensorflow.keras import callbacks

     model.compile(optimizer='adam', loss='binary_crossentropy',␣
      ↪metrics=['accuracy'])

     # Setup callbacks for model saving and early stopping
     checkpoint = callbacks.ModelCheckpoint('best_gru_model.h5',␣
      ↪monitor='val_accuracy', save_best_only=True, verbose=1)
     # EarlyStopping callback to stop training when the validation loss does not␣
      ↪improve
     early_stopping = callbacks.EarlyStopping(
         monitor='val_loss',     # Monitor model's validation loss
         patience=20,            # Patience is the number of epochs to wait after min␣
      ↪has been hit. After this number of no improvement, training stops.
         verbose=1,
         mode='min',             # The training will stop when the quantity monitored␣
      ↪has stopped decreasing
         restore_best_weights=True  # Restores model weights from the epoch with the␣
      ↪best value of the monitored quantity.
     )

     # Train the model
     history = model.fit(X_train_seq, y_train_seq,
                    validation_data=(X_val_seq, y_val_seq),
                    epochs=50,
                    batch_size=32,
                    verbose=1,
                    callbacks=[early_stopping]  # Including early stopping␣
↪callback here
                    )
```

```
Epoch 1/50
29/29 [==============================] - 6s 77ms/step - loss: 0.6925 - accuracy:
0.5291 - val_loss: 0.7072 - val_accuracy: 0.4470
Epoch 2/50
29/29 [==============================] - 0s 14ms/step - loss: 0.6852 - accuracy:
0.5587 - val_loss: 0.7075 - val_accuracy: 0.4603
Epoch 3/50
29/29 [==============================] - 0s 15ms/step - loss: 0.6807 - accuracy:
0.5631 - val_loss: 0.7168 - val_accuracy: 0.4404
Epoch 4/50
29/29 [==============================] - 0s 15ms/step - loss: 0.6657 - accuracy:
0.5982 - val_loss: 0.7284 - val_accuracy: 0.4702
Epoch 5/50
29/29 [==============================] - 0s 14ms/step - loss: 0.6345 - accuracy:
0.6454 - val_loss: 0.7468 - val_accuracy: 0.5298
Epoch 6/50
29/29 [==============================] - 0s 16ms/step - loss: 0.5976 - accuracy:
0.6839 - val_loss: 0.7796 - val_accuracy: 0.5166
Epoch 7/50
29/29 [==============================] - 0s 15ms/step - loss: 0.5515 - accuracy:
0.7091 - val_loss: 0.8115 - val_accuracy: 0.5132
Epoch 8/50
29/29 [==============================] - 1s 18ms/step - loss: 0.5167 - accuracy:
0.7102 - val_loss: 0.9595 - val_accuracy: 0.4934
Epoch 9/50
29/29 [==============================] - 1s 19ms/step - loss: 0.4900 - accuracy:
0.7344 - val_loss: 0.9190 - val_accuracy: 0.5066
Epoch 10/50
29/29 [==============================] - 1s 19ms/step - loss: 0.4441 - accuracy:
0.7695 - val_loss: 0.9767 - val_accuracy: 0.5099
Epoch 11/50
29/29 [==============================] - 1s 18ms/step - loss: 0.4281 - accuracy:
0.7728 - val_loss: 1.0525 - val_accuracy: 0.5199
Epoch 12/50
29/29 [==============================] - 0s 14ms/step - loss: 0.3880 - accuracy:
0.8145 - val_loss: 1.0836 - val_accuracy: 0.4934
Epoch 13/50
29/29 [==============================] - 0s 15ms/step - loss: 0.3565 - accuracy:
0.8266 - val_loss: 1.3027 - val_accuracy: 0.4868
Epoch 14/50
```

```
29/29 [==============================] - 0s 15ms/step - loss: 0.3310 - accuracy:
0.8430 - val_loss: 1.4210 - val_accuracy: 0.4967
Epoch 15/50
29/29 [==============================] - 0s 16ms/step - loss: 0.2924 - accuracy:
0.8639 - val_loss: 1.5185 - val_accuracy: 0.5099
Epoch 16/50
29/29 [==============================] - 0s 17ms/step - loss: 0.2705 - accuracy:
0.8749 - val_loss: 1.7421 - val_accuracy: 0.4934
Epoch 17/50
29/29 [==============================] - 1s 19ms/step - loss: 0.2490 - accuracy:
0.8880 - val_loss: 1.7424 - val_accuracy: 0.4868
Epoch 18/50
29/29 [==============================] - 1s 19ms/step - loss: 0.2405 - accuracy:
0.8858 - val_loss: 1.8908 - val_accuracy: 0.4570
Epoch 19/50
29/29 [==============================] - 1s 19ms/step - loss: 0.2159 - accuracy:
0.8979 - val_loss: 2.0068 - val_accuracy: 0.4868
Epoch 20/50
29/29 [==============================] - 0s 16ms/step - loss: 0.2009 - accuracy:
0.9012 - val_loss: 1.9645 - val_accuracy: 0.4768
Epoch 21/50
28/29 [===========================>..] - ETA: 0s - loss: 0.1840 - accuracy:
0.9118Restoring model weights from the end of the best epoch: 1.
29/29 [==============================] - 0s 15ms/step - loss: 0.1837 - accuracy:
0.9122 - val_loss: 2.0224 - val_accuracy: 0.4768
Epoch 21: early stopping
```

The `build_gru_model` function is designed to construct a Gated Recurrent Unit (GRU) network tailored to specific configurations, facilitating the exploration of different model architectures within our analysis. This function assembles a Sequential model beginning with a GRU layer that processes sequences based on the provided `time_steps` and feature size, allowing the model to maintain temporal information over longer sequences.

Following the initial recurrent layer, a Dropout layer is introduced to mitigate overfitting by randomly setting a fraction of the input units to 0 at each update during training time, which is controlled by `dropout_rate`. This helps in making the model robust and less prone to overfit statistical noise in the training data.

Another GRU layer is added without sequence retention (`return_sequences=False`), meaning this layer will only output the final result of the recurrent computation, reducing the sequence to a single vector that captures relevant temporal properties. This vector is then passed through another Dropout layer before reaching the output layer.

The output layer is a Dense layer with a single neuron and a sigmoid activation function to achieve a binary classification output, representing the probability of the input sequence belonging to one of two classes (e.g., positive or negative sentiment).

The model is compiled with the Adam optimizer and binary cross-entropy as the loss function, which are suitable for binary classification problems. This setup allows for flexible adjustments to the network's structure and can be easily extended or modified to experiment with different numbers of units, dropout rates, and sequence lengths.

```
[ ]: def build_gru_model(units, dropout_rate, time_steps):
         model = Sequential([
             GRU(units=units, return_sequences=True, input_shape=(time_steps,␣
      ↪X_train_seq.shape[2])),
             Dropout(dropout_rate),
             GRU(units=units, return_sequences=False),
             Dropout(dropout_rate),
             Dense(1, activation='sigmoid')
         ])
         model.compile(optimizer='adam', loss='binary_crossentropy',␣
      ↪metrics=['accuracy'])
         return model
```

This section of the code below systematically explores different configurations for the Gated Recurrent Unit (GRU) models to identify the optimal settings that maximize validation accuracy. Using a grid search strategy, the model is tested across a predefined range of hyperparameters.

The param_grid dictionary specifies the range of values for three critical hyperparameters: the number of units in each GRU layer (units), the dropout rate for regularization (dropout_rate), and the number of timesteps each input sequence contains (time_steps). These parameters are essential as they influence the model's ability to process and learn from the sequential data effectively. We evaluate models with varying complexities by altering these parameters to see how they affect performance, aiming to find a balance that minimizes overfitting while maintaining high accuracy.

For each combination of parameters in the grid, the process involves preparing the data into sequences that respect the specified `time_steps`, training a GRU model on these sequences, and evaluating its performance. This involves initializing a GRU model with the specified number of units and dropout rate for each parameter set using the `build_gru_model` function. The model is then trained on the training dataset with callbacks for early stopping and model checkpointing to mitigate overfitting and ensure only the best model according to validation accuracy is saved.

The validation accuracy for each model configuration is carefully monitored. By iterating over all possible combinations, the setup ensures that each model configuration is evaluated under the same conditions, allowing for an unbiased comparison of performance. The highest validation accuracy observed during the experiments dictates the selection of the best model configuration. This methodical testing and comparison process aids in selecting the most effective GRU model configuration that offers the best generalization on unseen data.

At the conclusion of this evaluation, the configuration yielding the highest validation accuracy is identified and outputted. This approach not only helps in pinpointing the best model parameters but also provides insights into how different settings impact the learning and general performance of GRU networks in processing time-series data.

```
[ ]: # Define parameter grid
     param_grid = {
         'units': [50, 100, 150],
         'dropout_rate': [0.1, 0.2, 0.3],
         'time_steps': [1, 2, 3]  # Including time_steps as a hyperparameter
}

best_overall_val_accuracy = 0
best_overall_config = None
best_model = None

for time_steps in param_grid['time_steps']:
    X_train_seq, y_train_seq = create_sequences(X_train.toarray(), y_train,␣
 ↪time_steps)
    X_val_seq, y_val_seq = create_sequences(X_val.toarray(), y_val, time_steps)
    X_test_seq, y_test_seq = create_sequences(X_test.toarray(), y_test,␣
 ↪time_steps)

    for units in param_grid['units']:
        for dropout in param_grid['dropout_rate']:
            print(f"Testing GRU with {units} units, {dropout} dropout, and␣
 ↪{time_steps} time steps.")

            model = build_gru_model(units=units, dropout_rate=dropout,␣
 ↪time_steps=time_steps)

            # Setup callbacks for early stopping and best model saving
            callback_list = [
                callbacks.EarlyStopping(monitor='val_loss', patience=10,␣
 ↪restore_best_weights=True),
                callbacks.
 ↪ModelCheckpoint(f'best_gru_model_{units}_{dropout}_{time_steps}.h5',␣
 ↪save_best_only=True)
            ]

            history = model.fit(X_train_seq, y_train_seq,␣
 ↪validation_data=(X_val_seq, y_val_seq),
                                epochs=50, batch_size=32,␣
 ↪callbacks=callback_list, verbose=0)

            val_accuracy = max(history.history['val_accuracy'])

            if val_accuracy > best_overall_val_accuracy:
                best_overall_val_accuracy = val_accuracy
                best_overall_config = (units, dropout, time_steps)
                best_model = model

print(f"Best GRU Configuration: Units={best_overall_config[0]},␣
 ↪Dropout={best_overall_config[1]}, Time Steps={best_overall_config[2]}, Val␣
 ↪Accuracy={best_overall_val_accuracy}")
```

```
Testing GRU with 50 units, 0.1 dropout, and 1 time steps.
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/engine/training.py:3103:
UserWarning: You are saving your model as an HDF5 file via `model.save()`. This
file format is considered legacy. We recommend using instead the native Keras
format, e.g. `model.save('my_model.keras')`.
  saving_api.save_model(
```

```
Testing GRU with 50 units, 0.2 dropout, and 1 time steps.
Testing GRU with 50 units, 0.3 dropout, and 1 time steps.
Testing GRU with 100 units, 0.1 dropout, and 1 time steps.
Testing GRU with 100 units, 0.2 dropout, and 1 time steps.
Testing GRU with 100 units, 0.3 dropout, and 1 time steps.
Testing GRU with 150 units, 0.1 dropout, and 1 time steps.
Testing GRU with 150 units, 0.2 dropout, and 1 time steps.
Testing GRU with 150 units, 0.3 dropout, and 1 time steps.
Testing GRU with 50 units, 0.1 dropout, and 2 time steps.
Testing GRU with 50 units, 0.2 dropout, and 2 time steps.
Testing GRU with 50 units, 0.3 dropout, and 2 time steps.
Testing GRU with 100 units, 0.1 dropout, and 2 time steps.
Testing GRU with 100 units, 0.2 dropout, and 2 time steps.
Testing GRU with 100 units, 0.3 dropout, and 2 time steps.
Testing GRU with 150 units, 0.1 dropout, and 2 time steps.
Testing GRU with 150 units, 0.2 dropout, and 2 time steps.
Testing GRU with 150 units, 0.3 dropout, and 2 time steps.
Testing GRU with 50 units, 0.1 dropout, and 3 time steps.
Testing GRU with 50 units, 0.2 dropout, and 3 time steps.
Testing GRU with 50 units, 0.3 dropout, and 3 time steps.
Testing GRU with 100 units, 0.1 dropout, and 3 time steps.
Testing GRU with 100 units, 0.2 dropout, and 3 time steps.
Testing GRU with 100 units, 0.3 dropout, and 3 time steps.
Testing GRU with 150 units, 0.1 dropout, and 3 time steps.
Testing GRU with 150 units, 0.2 dropout, and 3 time steps.
Testing GRU with 150 units, 0.3 dropout, and 3 time steps.
Best GRU Configuration: Units=100, Dropout=0.1, Time Steps=1, Val
Accuracy=0.5592105388641357
```

```
[ ]: def create_sequences(X, y, time_step):
         '''Function to reshape data for time series classification models (e..g,␣
      ↪LSTM, GRU, etc.)'''
         Xs, ys = [], []
         for i in range(len(X) - time_step):
             Xs.append(X[i:(i + time_step)])
             ys.append(y[i + time_step])
         return np.array(Xs), np.array(ys)
```

After identifying the best hyperparameters from our tuning process, the next step is to build and train the GRU model using these optimal settings.

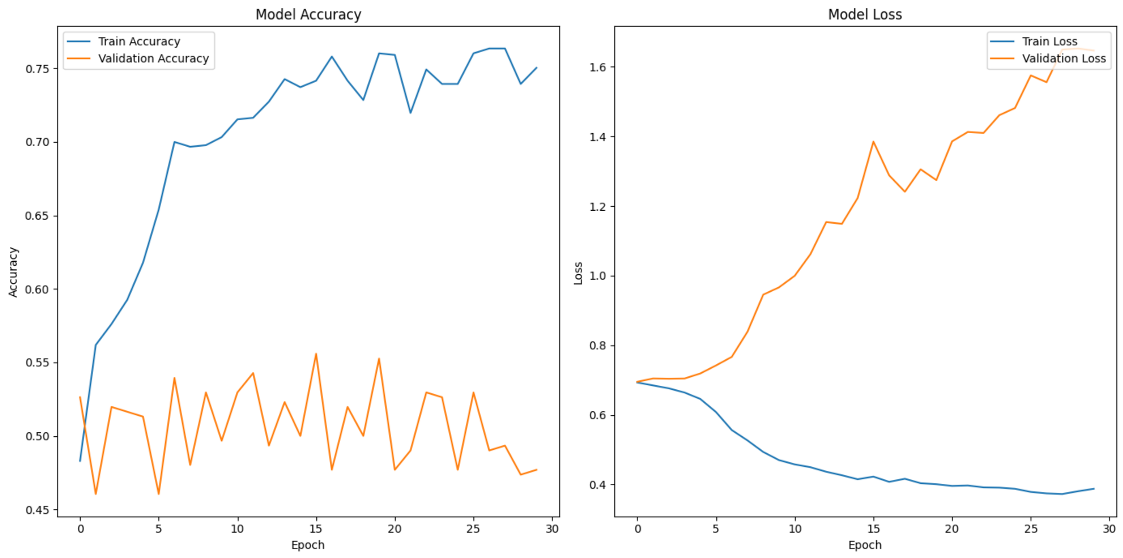
```
[ ]: best_time_steps = best_overall_config[2]
     X_train_seq_gru, y_train_seq_gru = create_sequences(X_train.toarray(), y_train,␣
      ↪best_time_steps)
     X_val_seq_gru, y_val_seq_gru = create_sequences(X_val.toarray(), y_val,␣
      ↪best_time_steps)
     X_test_seq_gru, y_test_seq_gru = create_sequences(X_test.toarray(), y_test,␣
      ↪best_time_steps)

     # Build and train the final model
     final_model = build_gru_model(units=best_overall_config[0],␣
      ↪dropout_rate=best_overall_config[1], time_steps=best_time_steps)
     history_gru = final_model.fit(X_train_seq_gru, y_train_seq_gru, epochs=30,␣
      ↪batch_size=32, validation_data=(X_val_seq_gru, y_val_seq_gru))
```

```
Epoch 1/30
29/29 [==============================] - 6s 43ms/step - loss: 0.6930 - accuracy:
0.4830 - val_loss: 0.6951 - val_accuracy: 0.5263
Epoch 2/30
29/29 [==============================] - 0s 16ms/step - loss: 0.6846 - accuracy:
0.5619 - val_loss: 0.7046 - val_accuracy: 0.4605
Epoch 3/30
29/29 [==============================] - 0s 16ms/step - loss: 0.6762 - accuracy:
0.5761 - val_loss: 0.7038 - val_accuracy: 0.5197
Epoch 4/30
29/29 [==============================] - 0s 14ms/step - loss: 0.6642 - accuracy:
0.5926 - val_loss: 0.7044 - val_accuracy: 0.5164
Epoch 5/30
29/29 [==============================] - 0s 13ms/step - loss: 0.6458 - accuracy:
0.6177 - val_loss: 0.7189 - val_accuracy: 0.5132
Epoch 6/30
29/29 [==============================] - 0s 13ms/step - loss: 0.6080 - accuracy:
0.6539 - val_loss: 0.7420 - val_accuracy: 0.4605
Epoch 7/30
29/29 [==============================] - 0s 14ms/step - loss: 0.5564 - accuracy:
0.6999 - val_loss: 0.7666 - val_accuracy: 0.5395
Epoch 8/30
29/29 [==============================] - 0s 13ms/step - loss: 0.5266 - accuracy:
0.6966 - val_loss: 0.8389 - val_accuracy: 0.4803
Epoch 9/30
29/29 [==============================] - 0s 14ms/step - loss: 0.4934 - accuracy:
0.6977 - val_loss: 0.9454 - val_accuracy: 0.5296
Epoch 10/30
29/29 [==============================] - 0s 13ms/step - loss: 0.4700 - accuracy:
0.7032 - val_loss: 0.9663 - val_accuracy: 0.4967
Epoch 11/30
29/29 [==============================] - 0s 13ms/step - loss: 0.4577 - accuracy:
0.7152 - val_loss: 0.9996 - val_accuracy: 0.5296
```

```
Epoch 12/30
29/29 [==============================] - 0s 13ms/step - loss: 0.4496 - accuracy:
0.7163 - val_loss: 1.0617 - val_accuracy: 0.5428
Epoch 13/30
29/29 [==============================] - 0s 14ms/step - loss: 0.4366 - accuracy:
0.7273 - val_loss: 1.1540 - val_accuracy: 0.4934
Epoch 14/30
29/29 [==============================] - 0s 13ms/step - loss: 0.4264 - accuracy:
0.7426 - val_loss: 1.1489 - val_accuracy: 0.5230
Epoch 15/30
29/29 [==============================] - 0s 12ms/step - loss: 0.4150 - accuracy:
0.7371 - val_loss: 1.2230 - val_accuracy: 0.5000
Epoch 16/30
29/29 [==============================] - 0s 14ms/step - loss: 0.4226 - accuracy:
0.7415 - val_loss: 1.3854 - val_accuracy: 0.5559
Epoch 17/30
29/29 [==============================] - 0s 16ms/step - loss: 0.4076 - accuracy:
0.7579 - val_loss: 1.2882 - val_accuracy: 0.4770
Epoch 18/30
29/29 [==============================] - 1s 20ms/step - loss: 0.4164 - accuracy:
0.7415 - val_loss: 1.2411 - val_accuracy: 0.5197
Epoch 19/30
29/29 [==============================] - 1s 21ms/step - loss: 0.4036 - accuracy:
0.7284 - val_loss: 1.3056 - val_accuracy: 0.5000
Epoch 20/30
29/29 [==============================] - 1s 20ms/step - loss: 0.4007 - accuracy:
0.7601 - val_loss: 1.2745 - val_accuracy: 0.5526
Epoch 21/30
29/29 [==============================] - 1s 28ms/step - loss: 0.3958 - accuracy:
0.7590 - val_loss: 1.3858 - val_accuracy: 0.4770
Epoch 22/30
29/29 [==============================] - 1s 24ms/step - loss: 0.3971 - accuracy:
0.7196 - val_loss: 1.4129 - val_accuracy: 0.4901
Epoch 23/30
29/29 [==============================] - 0s 17ms/step - loss: 0.3917 - accuracy:
0.7492 - val_loss: 1.4100 - val_accuracy: 0.5296
Epoch 24/30
29/29 [==============================] - 0s 17ms/step - loss: 0.3909 - accuracy:
0.7393 - val_loss: 1.4610 - val_accuracy: 0.5263
Epoch 25/30
29/29 [==============================] - 0s 16ms/step - loss: 0.3876 - accuracy:
0.7393 - val_loss: 1.4818 - val_accuracy: 0.4770
Epoch 26/30
29/29 [==============================] - 0s 14ms/step - loss: 0.3786 - accuracy:
0.7601 - val_loss: 1.5753 - val_accuracy: 0.5296
Epoch 27/30
29/29 [==============================] - 0s 13ms/step - loss: 0.3744 - accuracy:
0.7634 - val_loss: 1.5558 - val_accuracy: 0.4901
```

```
Epoch 28/30
29/29 [==============================] - 0s 15ms/step - loss: 0.3726 - accuracy:
0.7634 - val_loss: 1.6497 - val_accuracy: 0.4934
Epoch 29/30
29/29 [==============================] - 0s 16ms/step - loss: 0.3805 - accuracy:
0.7393 - val_loss: 1.6527 - val_accuracy: 0.4737
Epoch 30/30
29/29 [==============================] - 0s 16ms/step - loss: 0.3876 - accuracy:
0.7503 - val_loss: 1.6468 - val_accuracy: 0.4770
```

[ ]:
```
plot_history(history_gru)
```



Now, let's make a prediction on our test set and evaluate the performance of our LSTM implementation through visualizations.

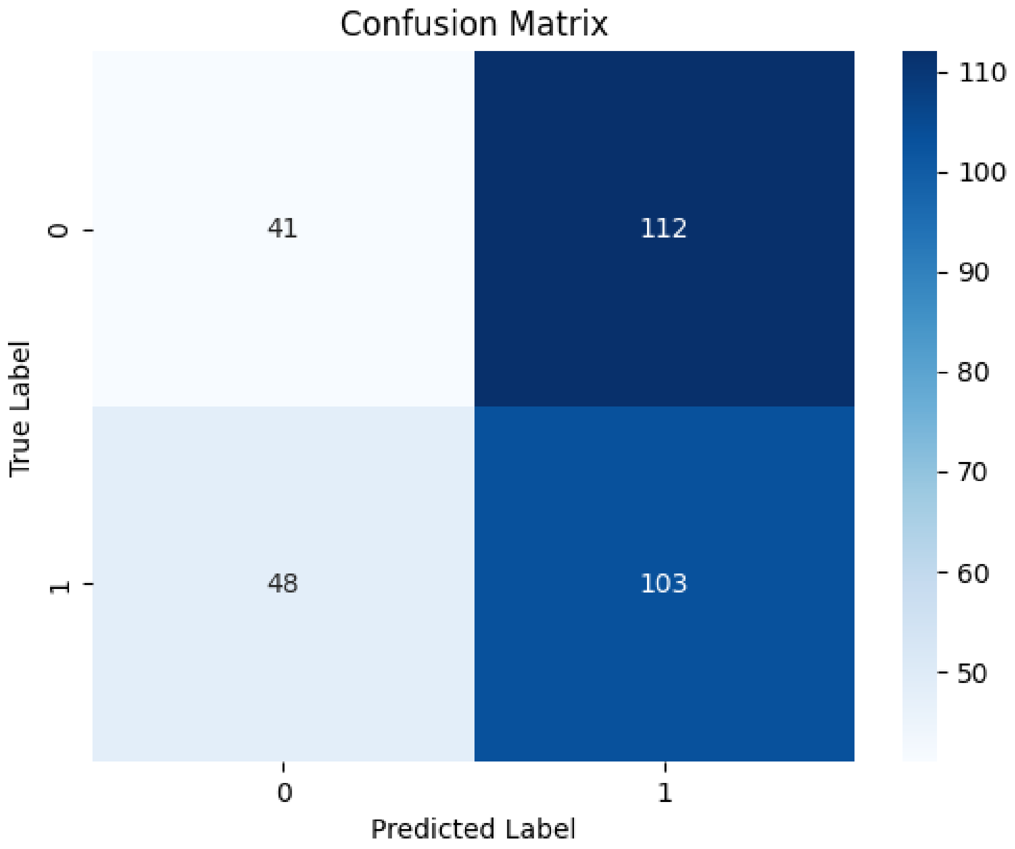
[ ]:
```
# Make predictions
y_pred_probs_gru = final_model.predict(X_test_seq_gru)
y_preds_gru = (y_pred_probs_gru > 0.5).astype(int)   # Convert probabilities to␣
↪binary output
```

```
10/10 [==============================] - 0s 9ms/step
```

[ ]:
```
from sklearn.metrics import confusion_matrix

# Classification report
report = classification_report(y_test_seq_gru, y_preds_gru)
print(report)

# Confusion Matrix
cm = confusion_matrix(y_test_seq_gru, y_preds_gru)
```

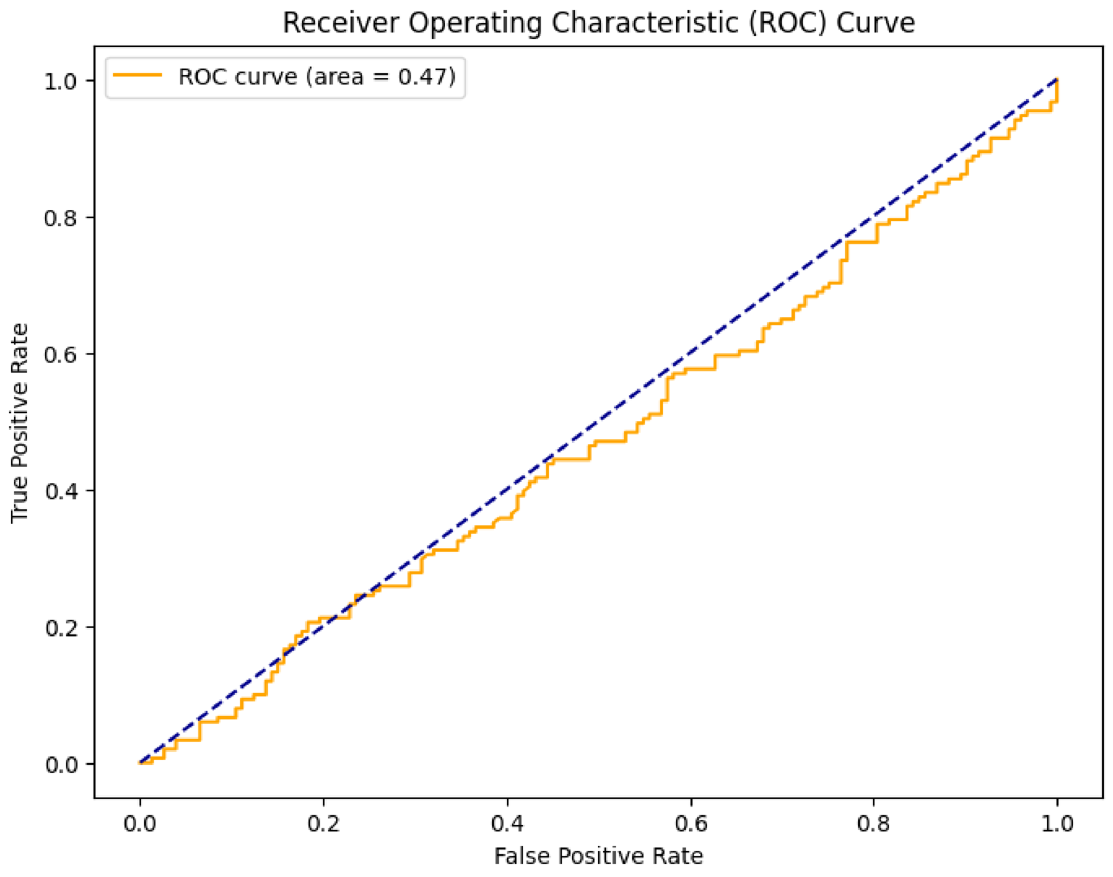
```
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.title('Confusion Matrix')
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.show()
```

```
              precision    recall  f1-score   support

           0       0.46      0.27      0.34       153
           1       0.48      0.68      0.56       151

    accuracy                           0.47       304
   macro avg       0.47      0.48      0.45       304
weighted avg       0.47      0.47      0.45       304
```



```
[ ]: from sklearn.metrics import roc_curve, roc_auc_score

     # Plot ROC Curve
     fpr, tpr, thresholds = roc_curve(y_test_seq_gru, y_pred_probs_gru)
```

```
# Calculate the AUC (Area under the ROC Curve)
roc_auc_gru = roc_auc_score(y_test_seq_gru, y_pred_probs_gru)

plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, color='orange', label=f'ROC curve (area = {roc_auc_gru:.
  ↪2f})')
plt.plot([0, 1], [0, 1], color='darkblue', linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend()
plt.show()
```



Discussion of Performance Results for GRU

The GRU model appears to suffer from overfitting based on the provided metrics. While the training accuracy steadily increases and reaches relatively high levels around 0.75, there is a considerable gap between the training and validation accuracy curves. The validation accuracy exhibits significant fluctuations and fails to achieve comparable levels, indicating that the model is learning patterns specific to the training data that do not generalize well to unseen instances.

The model loss graphs further reinforce this observation. The training loss decreases smoothly as the model becomes increasingly confident in its predictions on the training set. However, the validation loss follows an erratic pattern and ultimately trends upwards towards the end, which is a telltale sign of overfitting.

The confusion matrix and classification metrics provide additional insights into the model's performance. The confusion matrix reveals that the model correctly classified a reasonable number of instances for both classes (112 true positives and 41 true negatives), but it also misclassified a significant portion (103 false negatives and 48 false positives). The precision and recall values for both classes are modest, with the recall for class 0 being particularly low at 0.27, suggesting that the model struggles to accurately identify negative instances.

Furthermore, the Receiver Operating Characteristic (ROC) curve exhibits an area under the curve (AUC) of 0.47, which is relatively low and close to the performance of a random classifier (AUC = 0.5). This implies that the model's ability to distinguish between the positive and negative classes is suboptimal, further confirming the issues with generalization.

In summary, while the GRU model achieves reasonable training accuracy, it fails to effectively generalize this learning to the validation set, resulting in poor predictive performance on unseen data. This overfitting issue could potentially be mitigated by employing regularization techniques, introducing dropout layers, collecting more diverse training data, or exploring simpler architectures that are less prone to overfitting.

Extension Model 2.4 & 3.4 - Comparing Long Short-Term Memory Model with Gated Recurrent Unit Model In the universe of recurrent neural networks (RNNs), both Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) models stand out as robust solutions to the vanishing gradient problem, which hampers the training of traditional RNNs over long sequences. These models are pivotal in fields such as natural language processing, speech recognition, and time-series analysis due to their ability to remember information for long periods. However, despite their similarities, LSTMs and GRUs have distinct structural differences that confer unique advantages and disadvantages.

Long Short-Term Memory (LSTM) Models LSTMs are known for their complex internal structure, which includes three gates: input, forget, and output gates. This complexity allows them to perform exceptionally well on tasks that require understanding long-term dependencies. They can maintain information in memory for long durations, which is crucial for tasks where the context from early in the sequence is essential for making accurate predictions later on.

Advantages: - LSTMs excel in tasks where all-time slices are important thanks to their ability to preserve information for long periods through their intricate gating mechanisms.

Disadvantages: - The primary drawback of LSTMs is their computational complexity. The presence of three gates makes them parameter-heavy and computationally expensive to train, potentially leading to slower training times and greater resource consumption.

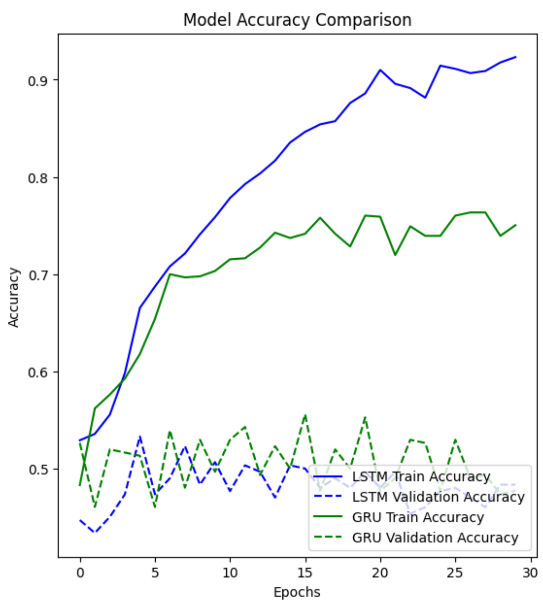
Gated Recurrent Unit (GRU) Models GRUs simplify the gating mechanism by combining the input and forget gates into a single "update gate" and merging the cell state and hidden state, resulting in a model that is easier to train and requires fewer computational resources.

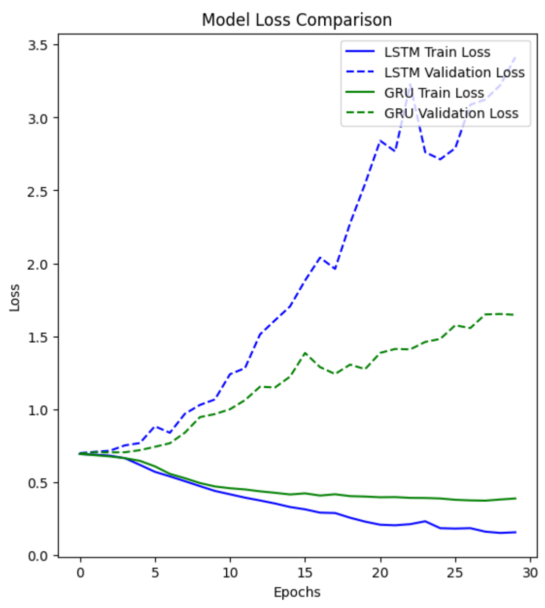
Advantages: - GRUs provide a more streamlined architecture, which can lead to faster training times and better performance on smaller or less complex datasets where the full power of LSTMs might not be necessary.

Disadvantages: - However, this simplification can be a drawback in scenarios that demand the nuanced capabilities of LSTMs, such as tasks with very long sequences or where the model needs to perform complex data adjustments across time steps.

Now, let's compare the performance of these models on our dataset.

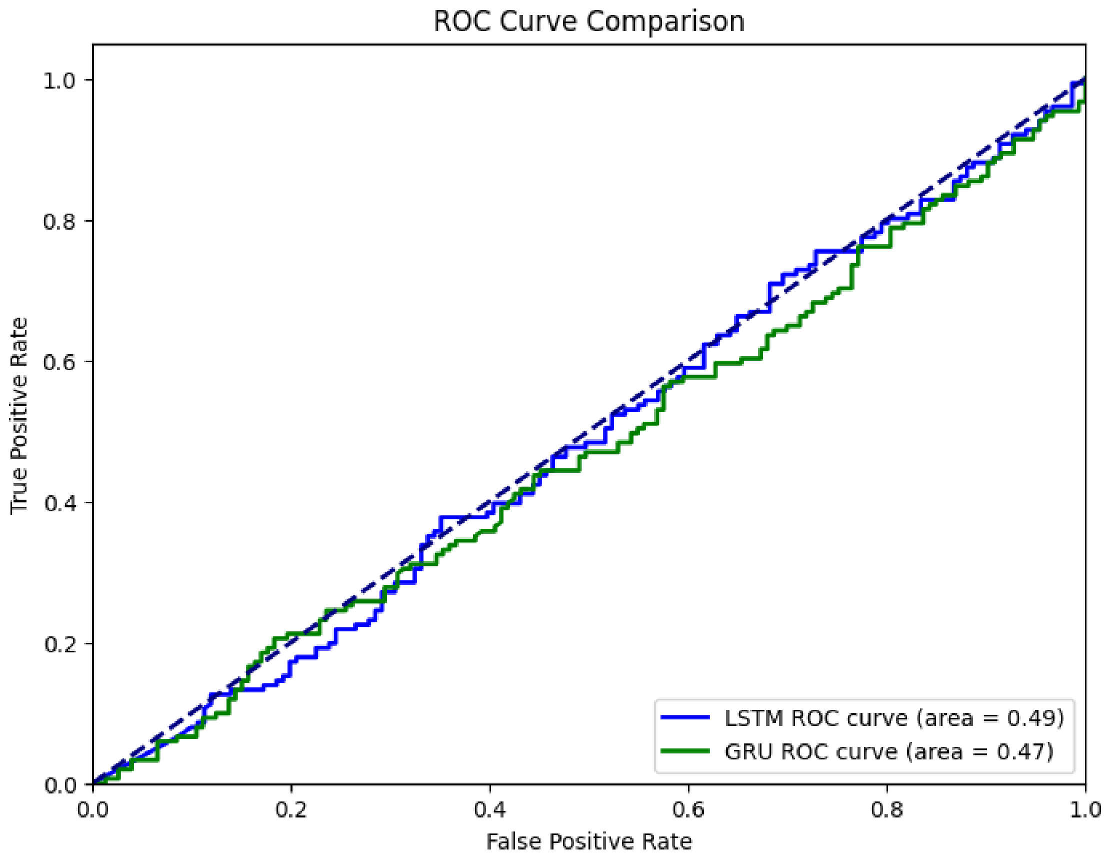
[ ]:
```
import matplotlib.pyplot as plt

def plot_combined_model_performance(history_lstm, history_gru):
    plt.figure(figsize=(14, 7))

    # Plotting Accuracy for both models
    plt.subplot(1, 2, 1)
    plt.plot(history_lstm.history['accuracy'], label='LSTM Train Accuracy',␣
 ↪color='blue')
    plt.plot(history_lstm.history['val_accuracy'], label='LSTM Validation␣
 ↪Accuracy', color='blue', linestyle='dashed')
    plt.plot(history_gru.history['accuracy'], label='GRU Train Accuracy',␣
 ↪color='green')
    plt.plot(history_gru.history['val_accuracy'], label='GRU Validation␣
 ↪Accuracy', color='green', linestyle='dashed')
    plt.title('Model Accuracy Comparison')
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')
    plt.legend(loc='lower right')

    # Plotting Loss for both models
    plt.subplot(1, 2, 2)
    plt.plot(history_lstm.history['loss'], label='LSTM Train Loss',␣
 ↪color='blue')
    plt.plot(history_lstm.history['val_loss'], label='LSTM Validation Loss',␣
 ↪color='blue', linestyle='dashed')
    plt.plot(history_gru.history['loss'], label='GRU Train Loss', color='green')
    plt.plot(history_gru.history['val_loss'], label='GRU Validation Loss',␣
 ↪color='green', linestyle='dashed')
    plt.title('Model Loss Comparison')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend(loc='upper right')

    plt.show()

# Call the function with both histories
plot_combined_model_performance(history_lstm, history_gru)
```





[ ]:
```
import matplotlib.pyplot as plt
from sklearn.metrics import roc_curve, auc

def plot_combined_roc_curve(y_test_lstm, y_pred_probs_lstm, y_test_gru,␣
 ↪y_pred_probs_gru):
    # Calculate ROC curve and ROC area for each model
    fpr_lstm, tpr_lstm, _ = roc_curve(y_test_lstm, y_pred_probs_lstm)
    roc_auc_lstm = auc(fpr_lstm, tpr_lstm)

    fpr_gru, tpr_gru, _ = roc_curve(y_test_gru, y_pred_probs_gru)
    roc_auc_gru = auc(fpr_gru, tpr_gru)

    # Plotting
    plt.figure(figsize=(8, 6))
    plt.plot(fpr_lstm, tpr_lstm, color='blue', lw=2, label=f'LSTM ROC curve␣
 ↪(area = {roc_auc_lstm:.2f})')
    plt.plot(fpr_gru, tpr_gru, color='green', lw=2, label=f'GRU ROC curve (area␣
 ↪= {roc_auc_gru:.2f})')
    plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('ROC Curve Comparison')
    plt.legend(loc="lower right")
    plt.show()
```

```
plot_combined_roc_curve(y_test_seq_lstm, y_pred_probs_lstm, y_test_seq_gru,␣
  ↪y_pred_probs_gru)
```



Comparing the Performance Results for LSTM and GRU

Accuracy and Loss: - The LSTM model achieves higher training accuracy compared to the GRU model, indicating better learning of patterns in the training data. - However, the validation accuracy of the LSTM fluctuates more compared to the GRU, suggesting potential overfitting issues. - The LSTM has lower training loss compared to the GRU, but its validation loss is higher and shows more variation, further reinforcing the overfitting concerns.

ROC Curve and AUC: - Both models have relatively low AUC scores, with the LSTM slightly better at 0.49 compared to the GRU's 0.47. - An AUC of 0.5 represents a random classifier, so both models are performing only marginally better than random guessing in distinguishing between the two classes. - The ROC curves for both models are quite similar, with the LSTM's curve slightly higher than the GRU's, indicating slightly better performance in balancing true positive and false positive rates.

Overall, while the LSTM model achieves higher training accuracy and lower training loss, it also exhibits more significant overfitting issues, as evidenced by the larger gap between training and validation metrics. The GRU model, although performing slightly worse in terms of training metrics, seems to generalize better to the validation data, with less fluctuation in validation accuracy and loss.

However, both models struggle to effectively distinguish between the two classes, as indicated by the low AUC scores close to 0.5. The LSTM has a slight edge over the GRU in terms of AUC, but the difference is marginal.

In summary, the LSTM appears to have an advantage in learning the training data patterns, but it suffers from more pronounced overfitting issues. The GRU, while not performing as well on the training data, exhibits better generalization to unseen data. Ultimately, both models require further improvements to achieve better separability between the classes and improve their predictive performance.

1.9.8 Extension Models Comparison - TSF, LSTM, and GRU

Conceptual Framework In search of the most effective model for time-series analysis, we have implemented three significant models: Time Series Forests (TSF), Long Short-Term Memory (LSTM) units, and Gated Recurrent Units (GRU). Each of these models offers unique advantages and challenges in processing and predicting time-series data, making them suitable for different types of problems.

Time Series Forest (TSF) **Time Series Forests**, as detailed in the resources in our references, are an ensemble learning technique based on decision trees. Specifically, TSF uses a collection of decision trees to capture the time-series patterns and dynamics effectively. The model's strength lies in its ability to provide interpretable models that are relatively simple to understand and implement.

Advantages: - TSF is very effective for feature-based time-series analysis, where the features effectively capture the underlying patterns. - The model offers good interpretability compared to LSTM and GRU, as the decision paths in trees can be easily visualized and understood.

Disadvantages: - However, TSF might struggle with very long sequences or where the temporal dependencies extend across lengthy intervals, as traditional feature engineering might not capture long-term dependencies effectively.

Long Short-Term Memory (LSTM) **LSTMs** are designed to avoid the long-term dependency problem typical of standard recurrent neural networks (RNNs). They achieve this through their sophisticated gate mechanisms which regulate the flow of information.

Advantages: - Excellent for applications where the sequence length is extensive and maintaining information across this length is crucial for performance. - Capable of learning order dependence in sequence prediction problems, making them superior for complex problems involving multiple time steps ahead prediction.

Disadvantages: - LSTMs come with a higher computational and model complexity cost due to their complex architectures, potentially leading to longer training times and requiring more data to train effectively.

Gated Recurrent Unit (GRU) **GRUs** are a streamlined alternative to LSTMs, designed to solve the same problems without the complexity of multiple gates.

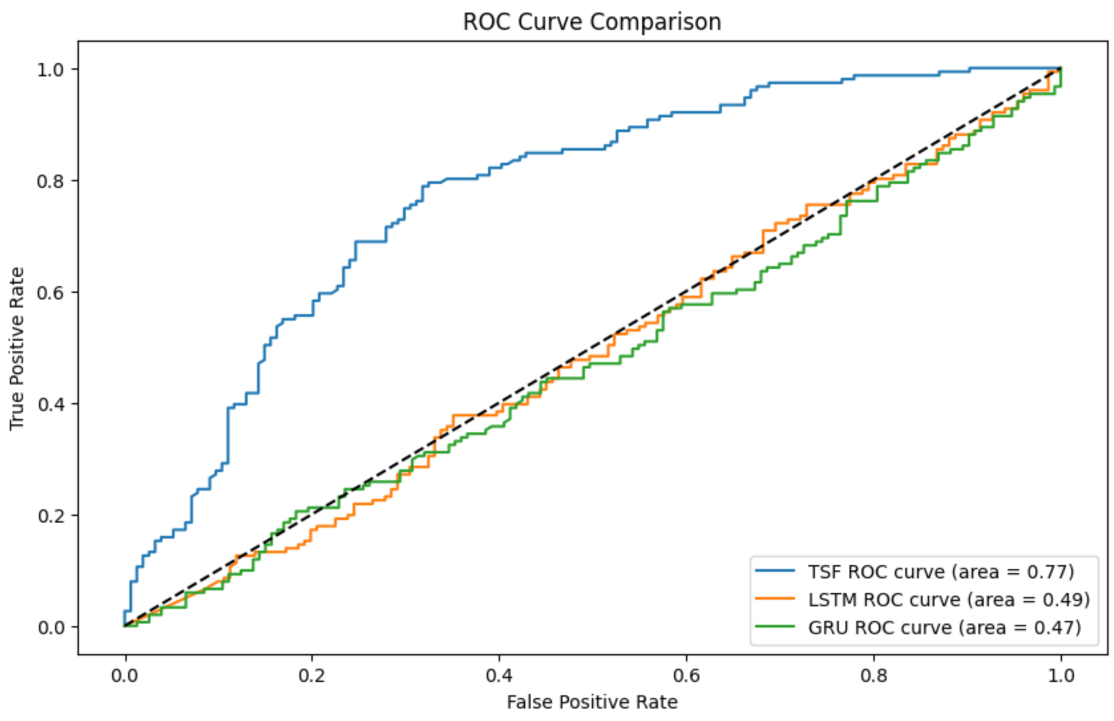
Advantages: - GRUs simplify the model architecture by combining several gates into one or two, reducing the computational overhead and making them faster to train than LSTMs. - They are particularly effective in cases where the sequence length isn't excessively long, which allows for quicker learning and adaptation.

Disadvantages: - The simplification can lead to a loss in the ability to precisely model complex dependencies compared to LSTMs, particularly in tasks that benefit from intricate gating mechanisms to manage the information flow.

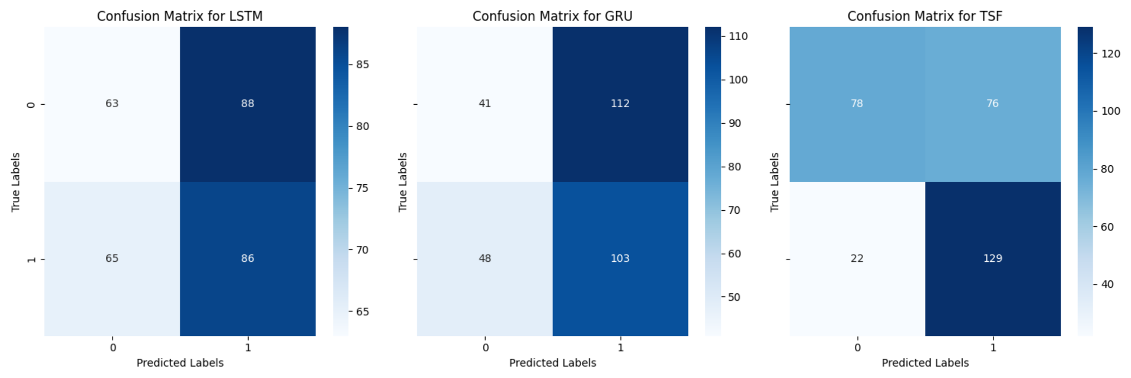
Making the Decision Selecting the appropriate model—TSF, LSTM, or GRU—will depend on the particular demands and limitations of our time-series analysis task. TSF is a good choice for less complex issues or when it's crucial to easily interpret the model's decisions. On the other hand, LSTMs are better suited for intricate problems involving long sequences, where the benefits in model accuracy outweigh the higher computational demands. Meanwhile, GRUs provide a middle ground, offering a less complex solution that doesn't compromise too much on performance and is ideal for moderately complex tasks that require relatively faster training. Since our dataset was limited in this project, let's review and compare the performance metrics of these three models and interpret the results.

Performance Comparison Among TSF, LSTM, and GRU

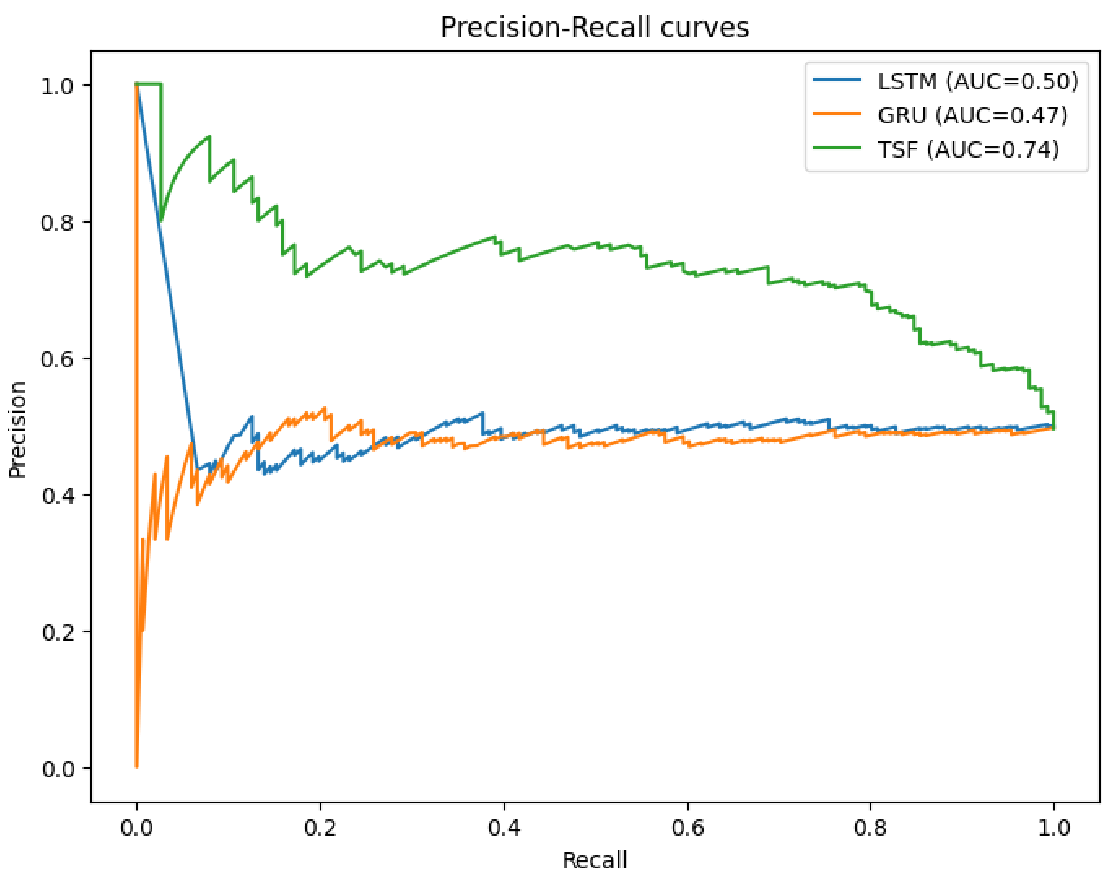
[ ]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import roc_curve, auc, roc_auc_score

def plot_combined_roc_curve(y_tests, y_pred_probs, labels):
    plt.figure(figsize=(10, 6))
    for y_test, y_pred_prob, label in zip(y_tests, y_pred_probs, labels):
        fpr, tpr, _ = roc_curve(y_test, y_pred_prob)
        roc_auc = auc(fpr, tpr)
        plt.plot(fpr, tpr, label=f'{label} ROC curve (area = {roc_auc:.2f})')
    plt.plot([0, 1], [0, 1], 'k--')
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('ROC Curve Comparison')
    plt.legend(loc="lower right")
    plt.show()

# Function to plot ROC Curves for all models
plot_combined_roc_curve(
    [y_test, y_test_seq_lstm, y_test_seq_gru],
    [y_pred_proba_tsf, y_pred_probs_lstm.flatten(), y_pred_probs_gru.flatten()],
    ['TSF', 'LSTM', 'GRU']
)
```



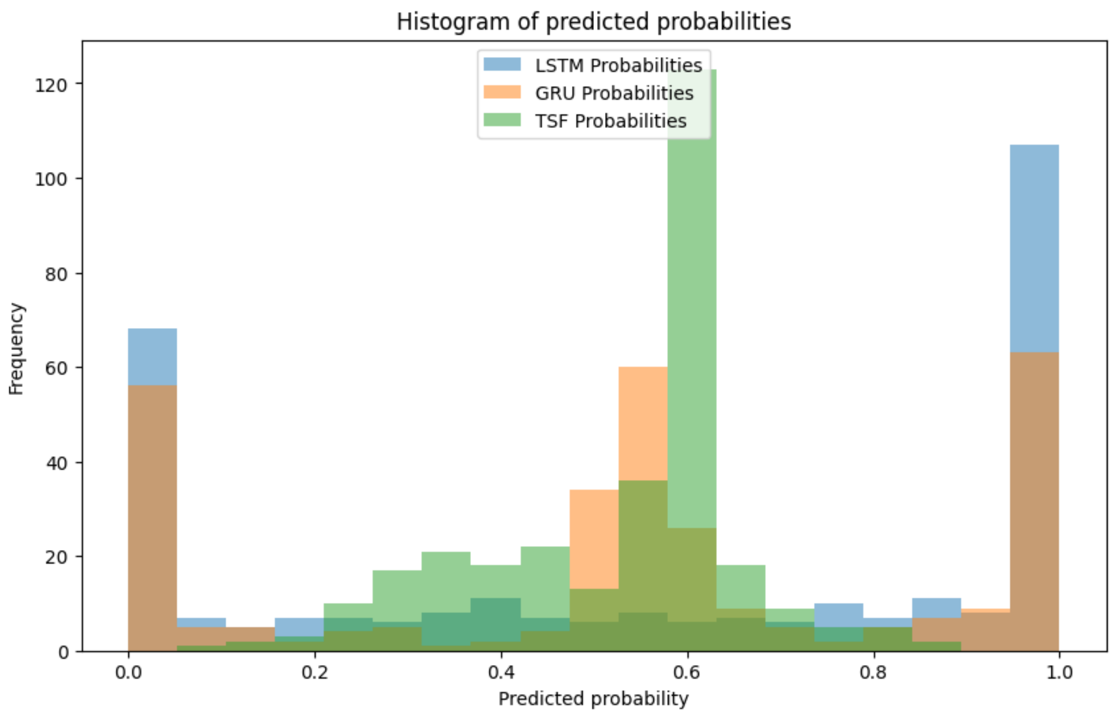
```
[ ]: import matplotlib.pyplot as plt
     import seaborn as sns
     from sklearn.metrics import confusion_matrix

     def plot_confusion_matrices(y_true, y_preds, labels):
         fig, axes = plt.subplots(nrows=1, ncols=len(labels), figsize=(15, 5),␣
      ↪sharey=True)
         for idx, (y_pred, label) in enumerate(zip(y_preds, labels)):
             cm = confusion_matrix(y_true[idx], y_pred)
             sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', ax=axes[idx])
             axes[idx].set_title(f'Confusion Matrix for {label}')
             axes[idx].set_xlabel('Predicted Labels')
             axes[idx].set_ylabel('True Labels')
         plt.tight_layout()
         plt.show()

     # Assuming y_pred_lstm, y_pred_gru, y_pred_tsf are your predictions
     plot_confusion_matrices(
         [y_test_seq_lstm, y_test_seq_gru, y_test],
         [y_preds_lstm, y_preds_gru, y_pred],
         ['LSTM', 'GRU', 'TSF']
     )
```



```
[ ]: from sklearn.metrics import precision_recall_curve

     def plot_precision_recall_curves(y_tests, y_scores, model_labels):
         plt.figure(figsize=(8, 6))
         for y_test, y_score, label in zip(y_tests, y_scores, model_labels):
             precision, recall, _ = precision_recall_curve(y_test, y_score)
             plt.plot(recall, precision, label=f'{label} (AUC={auc(recall,␣
      ↪precision):.2f})')

         plt.xlabel('Recall')
         plt.ylabel('Precision')
         plt.title('Precision-Recall curves')
         plt.legend()
         plt.show()

     # Example call to the function
     plot_precision_recall_curves(
         [y_test_seq_lstm, y_test_seq_gru, y_test],
         [y_pred_probs_lstm.flatten(), y_pred_probs_gru.flatten(), y_pred_proba_tsf],
         ['LSTM', 'GRU', 'TSF']
     )
```



[ ]:
```
def plot_probability_histograms(y_scores, labels):
    plt.figure(figsize=(10, 6))
    bins = np.linspace(0, 1, 20)
    for y_score, label in zip(y_scores, labels):
        plt.hist(y_score, bins, alpha=0.5, label=f'{label} Probabilities')
    plt.xlabel('Predicted probability')
    plt.ylabel('Frequency')
    plt.legend(loc='upper center')
    plt.title('Histogram of predicted probabilities')
    plt.show()

plot_probability_histograms(
    [y_pred_probs_lstm.flatten(), y_pred_probs_gru.flatten(), y_pred_proba_tsf],
    ['LSTM', 'GRU', 'TSF']
)
```



Comparing the Performance Results for TSF, LSTM, and GRU

From the **ROC curve**, we observe that TSF has the highest area under the curve (AUC), indicating superior performance in distinguishing between the positive and negative classes. The ROC AUC is a robust performance metric, unaffected by the class imbalance, and a higher AUC implies better model performance.

- **TSF's Success**: The superior performance of TSF suggests that the features extracted are quite effective for this specific time-series problem. The decision trees within TSF could be capturing essential patterns in the data without overfitting, benefiting from both the robustness of ensemble methods and the interpretability of decision trees.
- **LSTM's Underperformance**: LSTM's AUC is considerably lower. This could be due to overfitting, insufficient training data, or a mismatch between the LSTM's capability to model long-term dependencies and the data's characteristics, which might not require such complex temporal modeling.
- **GRU's Similarity to LSTM**: GRU's performance is similar to that of LSTM, slightly better, but still below TSF. Given that GRU is a simplified version of LSTM, it is possible that the dataset does not have long-term dependencies that would benefit from LSTM's complex gating mechanisms, making GRU's simplifications more suited but still not optimal compared to TSF.

The **confusion matrices** provide insights into the type I (false positives) and type II (false negatives) errors each model makes.

- **High False Positives for LSTM and GRU**: Both LSTM and GRU models have higher false positives compared to TSF. This can be tied to the recurrent nature of these networks, which might be sensitive to noise and thus overfitting to the noise, predicting false positives more frequently.

- **TSF's Balanced Performance**: TSF shows a more balanced performance, with fewer false positives and false negatives. This suggests that the features used by TSF are robust and generalize well on the test data.

The **Precision-Recall (PR)** curves focus on the performance of the positive class and are particularly useful for evaluating models on imbalanced datasets.

- **TSF's Superior Precision**: The PR curve shows TSF with a higher AUC, indicating both high precision and recall. This implies that when TSF predicts an instance as positive, it is correct more often than not and it also captures a large proportion of the actual positives.
- **GRU and LSTM's Lower Precision**: LSTM and GRU show lower precision, which can be associated with their higher false-positive rates as seen in the confusion matrices.

This **histogram** reveals the distribution of predicted probabilities for each class by each model.

- **TSF's Discriminative Power**: TSF seems to have better discriminative power as the probabilities are more polarized towards the extremes, indicating higher confidence in its predictions.
- **LSTM and GRU's Uncertainty**: LSTM and GRU show more predictions with intermediate probabilities, which could imply a degree of uncertainty in the model's predictions.

Overall,

- **TSF's Edge**: The superior performance of TSF in this context may arise from its capability to effectively capture time-dependent patterns with a simpler model and without the need for complex temporal dependency modeling, which can be an advantage when the patterns are more straightforward or the dataset is not extensive enough to train more complex models like LSTM or GRU.
- **RNNs' Struggle**: LSTM and GRU might struggle due to the complexity of their architectures, which are not fully leveraged in this context. Additionally, these models might require more data to generalize well and avoid overfitting, which is indicated by their relatively lower precision and higher false-positive rates.

To conclude,

- For datasets where the key features can be extracted and do not involve complex long-term dependencies, **TSF** would be the preferred model due to its performance and interpretability.
- **LSTM and GRU** could be considered for more complex sequences where their ability to model long-term dependencies can be harnessed. However, care must be taken to provide sufficient data and apply regularization techniques to prevent overfitting, as suggested by their overall lower performance metrics in the provided graphs.

1.10 Section 10: Discussion of Conclusions (Final Pipeline)

Finally, let's compare all the models we worked with in this end-to-end pipeline, and try to draw conlusions on which one we might opt for in which case.

Metric	BLR	XGBoost	SVM	TSF	LSTM	GRU
Test Accuracy	0.6787	0.682	0.6852	0.6787	0.49	0.47
Precision (Class 0)	0.75	0.77	0.76	0.7800	0.49	0.46
Recall (Class 0)	0.54	0.55	0.54	0.5100	0.42	0.27
F1-Score (Class 0)	0.63	0.63	0.63	0.6100	0.45	0.34
Precision (Class 1)	0.64	0.64	0.64	0.6300	0.49	0.48
Recall (Class 1)	0.82	0.83	0.82	0.8500	0.57	0.68
F1-Score (Class 1)	0.72	0.72	0.72	0.7200	0.53	0.56

Acknowledgement of Limited Data:

In this project, the amount of data available for training the models is a significant factor that can affect performance. Limited data can lead to models not capturing the complexity of the problem, potentially impacting their ability to generalize to new, unseen data. This limitation is crucial when evaluating model performance as it provides context for understanding the results.

Model Architectures and Potential Overfitting:

The BLR, XGBoost, and SVM models approach the mood prediction task from a feature-based perspective, likely not accounting for temporal dependencies. In contrast, TSF, LSTM, and GRU are time-series models designed to capture sequential information. Time-series models, especially LSTM and GRU, are sophisticated in their parameterization, allowing them to potentially model complex dependencies in sequence data. However, their architecture complexity also increases the risk of overfitting, particularly with limited data. Overfitting might be a reason for the lower test accuracy observed in LSTM and GRU compared to their simpler counterparts.

Differences Between Sequential Models and Other Models:

Sequential models like LSTM and GRU have mechanisms to remember information from previous data points, which is crucial for time-series prediction tasks. The TSF model also takes time into account but does so by transforming the time-series data into a feature space that standard classifiers can use. BLR, XGBoost, and SVM do not inherently consider the sequence in which data points occur, possibly leading to underperformance if the sequence significantly influences the outcome.

Scenarios Where Different Models Might Come Handy:

Considering different scenarios, BLR, XGBoost, and SVM have shown commendable performance in terms of accuracy and F1-Scores, especially in recognizing Class 1 (presumably 'Positive' mood), which can be more straightforward scenarios where the sequence of events may not play a significant role. In contrast, time-series models like TSF, LSTM, and GRU would be more suitable for scenarios where historical context and the sequence of events critically influence the prediction outcome.

Metric-Based Performance:

When comparing the performance, the traditional models (BLR, XGBoost, SVM) performed similarly and had relatively high accuracy scores, suggesting that they could capture the essence of the mood states to some extent without considering temporal dependencies. This might indicate that the mood states in the dataset were influenced more by the content of the calendar events rather than their sequence.

The TSF model, while still being a time-series model, had an accuracy comparable to the traditional models. This might be due to its feature-based approach to time-series data, which may bridge the gap between traditional models and more complex time-series models. Its relatively high recall for Class 1 suggests it can predict 'Positive' moods effectively, possibly due to capturing certain temporal patterns without the complexity of LSTM or GRU models.

The LSTM and GRU models had lower performance metrics across the board, which might be attributed to their complex architectures. Given the limited dataset, they may have struggled to learn generalized patterns, leading to potential overfitting. However, the recall for Class 1 is higher for GRU than LSTM, indicating some capacity to identify 'Positive' mood states, albeit less precisely.

In different scenarios, if the prediction task is not heavily reliant on understanding the sequence of events or when data is scarce, traditional models might be more appropriate. However, if there's an abundance of sequential data and the task demands understanding the progression of events over time, time-series models, despite their current performance, would be more advantageous. For such scenarios, investing in collecting more data, feature engineering to reduce dimensionality, and employing regularization techniques to mitigate overfitting would be crucial steps to improve the LSTM and GRU models' performance.

1.11 Section 11: Executive Summary - Mood Prediction Based on Calendar Events

1.11.1 First Pipeline, Second Pipeline, and Final Pipeline

1.11.2 1- Executive Summary from the First Pipeline

1.11.3 Project Overview

This project embarked on the ambitious task of predicting daily mood from calendar events using a dataset extracted from a Google Calendar associated with a school email address, spanning from June 29, 2021 to February 15, 2024. The analysis delved into personal engagements, academic deadlines, and other activities to discern patterns influencing mood fluctuations.

1.11.4 Methodology

The methodological framework of this project consisted of several structured stages:

1- Data Acquisition and Preprocessing Data was gathered via Google Takeout, providing an extensive history of calendar events. Careful sampling resulted in a dataset where each event's features, such as date, duration, and attendees, were recorded. Ethical considerations were paramount, ensuring that the data was personal yet non-sensitive. Through rigorous data ingestion, transformation, and cleaning processes, including the removal of duplicates and incomplete records, the dataset was curated to accurately reflect daily schedules. Features were engineered to encapsulate temporal elements, such as the start and end times of events, leading to an aggregated daily summary. These were then manually labeled with mood categories based on the nature of the events and personal recollections of the day's emotional impact.

2- Exploratory Analysis Various visual analyses, including histograms, bar charts, and scatter plots, were conducted to understand the distribution of events and their relationship with the labeled moods. Insights from these analyses informed the selection of features for modeling and revealed patterns in how certain types of days tend to correlate with specific moods.

3- Model Development A Multinomial Logistic Regression (MLR) model was chosen for its suitability for multi-class prediction and interpretability. The dataset was divided into training, validation, and test sets, with 60%, 20%, and 20% splits, respectively. The model training involved cross-validation and hyperparameter tuning using GridSearchCV to optimize performance metrics like accuracy, precision, recall, and F1-score.

4- Model Evaluation The model's ability to predict mood from calendar data was quantified using confusion matrices, feature importance charts, and ROC curves with AUC scoring. While the model excelled in identifying 'Relaxed' and 'Overwhelmed' moods, it faced difficulties in distinguishing between 'Busy' and 'Productive' moods. 'Average duration of events' emerged as the most influential predictor, with AUC scores for 'Relaxed' and 'Overwhelmed' being particularly high, indicative of the model's effective differentiation capabilities.

1.11.5 Visualization of Results

Key visualizations include:

- Confusion Matrix: Demonstrated the model's predictive accuracy and areas of misclassification.
- Feature Importance Chart: Identified 'avg_duration' as a significant predictor of mood.
- ROC Curves: Illustrated the model's capability to distinguish between the moods across different classification thresholds.

1.11.6 Key Findings

The analyses revealed the following key insights:

- Temporal Patterns and Mood Correlation: The analysis revealed that certain times of the week and specific event durations were consistently related to particular moods. For example, weekdays loaded with meetings and deadlines often led to 'Busy' or 'Overwhelmed' labels, while weekends with fewer and more leisurely events corresponded to 'Relaxed' moods. This temporal pattern provides a foundation for predicting mood swings and offers a strategic approach to personal scheduling that could enhance overall well-being.
- Event Duration as a Mood Indicator: The project highlighted the average duration of events as a significant predictor of mood. Longer events, mainly clustered together, indicated 'Overwhelmed' days, while shorter and well-spaced events aligned with 'Relaxed' and 'Productive' moods. This insight suggests that the number of events and their length and distribution throughout the day are crucial determinants of daily emotional states.
- Model's Predictive Strengths: The Multinomial Logistic Regression model demonstrated robust predictive strength for days marked as 'Relaxed' or 'Overwhelmed.' The high AUC scores for these moods confirm the model's effectiveness in distinguishing days characterized by clear-cut emotional outcomes, providing a reliable framework for mood prediction.
- Challenges in Nuanced Differentiation: While the model succeeded in differentiating between moods with distinct emotional states, it struggled with nuanced classification between 'Busy' and 'Productive' moods. This challenge underscores the complexity of human emotions and

the subtleties involved in their prediction, calling for more sophisticated modeling techniques or additional context-specific data to improve accuracy.

- Practical Implications for Personal Scheduling: The insights gained from the model's performance could inform personal and organizational scheduling tools. By understanding the relationship between scheduled events and mood, individuals and teams could plan activities that promote productivity while mitigating stress and burnout.
- Potential for Personalized Well-being Tools: The project's findings lay the groundwork for personalized well-being applications. By integrating calendar data with mood predictions, technology could potentially offer proactive suggestions for schedule adjustments that cater to an individual's mental health and productivity preferences.
- Shortcomings in Model Interpretability: Despite the interpretability of the Logistic Regression model, certain aspects, such as the importance and interaction of features, require deeper exploration. For instance, the interplay between event duration and start times could be further examined to understand their compound effect on mood prediction.

1.11.7 2- Executive Summary from the Second Pipeline

1.11.8 Key Changes Since First Pipeline

In the second iteration of my mood prediction analysis, I've introduced significant refinements to bolster my model's predictive accuracy and to gain deeper insights into the relationship between calendar events and mood states. Notably, I implemented a series of methodological adjustments and conducted additional explorations, as detailed below:

- **Advanced Feature Engineering**: Leveraging the insights gained from my first pipeline, I dove deeper into feature engineering, focusing on the nuances of temporal patterns and the intricacies of event descriptions. By synthesizing more sophisticated attributes by combining numeric and tokenized textual data, I aimed to capture subtle indicators of mood variations. Additionally, I paid attention to implementing all the data augmentation suggestions that I received as feedback to my first pipeline.

- **Enhanced Model Selection**: While the initial project utilized Multinomial Logistic Regression, I expanded my model suite to include XGBoost and Support Vector Machine algorithms. This strategic choice was driven by the need to explore different learning paradigms – ensemble methods with XGBoost and high-dimensional mapping with SVM – to improve the nuance in my predictions.

- **Hyperparameter Optimization**: I undertook a rigorous hyperparameter tuning process, employing tools such as GridSearchCV and RandomizedSearchCV to meticulously calibrate my models. This was done to ensure that I harness the full potential of the selected algorithms.

- **Model Comparison and Selection**: Armed with a set of diverse models, I meticulously evaluated their performance through an array of metrics, enabling me to perform a comparative analysis. This helped in identifying the most suitable model that strikes an optimal balance between precision and recall.

- **Pipeline Visualization and Interpretation**: Throughout this iteration, I placed significant emphasis on visualizing my pipeline stages and model performances. From precision-recall curves to feature importances, each visualization was carefully selected to elucidate the strengths and weaknesses of our approach.

1.11.9 Project Overview

In the second phase of my project, I've taken strides to refine mood prediction from calendar data by introducing more complex algorithms and incorporating dimensionality reduction. I've expanded my approach to include Binary Logistic Regression, XGBoost, and SVM, delving deeper into the subtleties of mood classification. By applying PCA, I've extracted the essence of my dataset, ensuring that my models focus on the most influential features. The thorough hyperparameter optimization conducted enhances the precision of my predictions, offering a nuanced view of the relationship between daily activities and emotional states.

1.11.10 Methodology Update in Second Pipeline

In the development of my second pipeline, I delved into the data-driven exploration of mood prediction from calendar events by incorporating enhanced modeling techniques and applying dimensionality reduction to streamline features.

1- Data Refinement I maintained the rigorous data preprocessing standards set in my first pipeline, ensuring the data's integrity and relevance. This included meticulous cleaning, transformation, and the engineering of new features to encapsulate the nuanced interplay between daily activities and emotional states.

2- Advanced Analytical Techniques Leveraging deeper exploratory analysis, I scrutinized the refined dataset for complex patterns and relationships that might influence mood predictions. This step involved examining new visualizations and statistical measures to better understand the data's structure and the predictive power of various features.

3- Expanding Model Horizons I expanded the suite of predictive models beyond MLR to include Binary Logistic Regression, XGBoost, and SVM—each subject to a thorough process of cross-validation and hyperparameter tuning. This broadened perspective enabled a comparative performance evaluation across diverse algorithmic approaches.

4- Enhanced Model Evaluation With a focus on precision, I used a refined set of evaluation metrics including updated confusion matrices, ROC curves, and precision-recall analyses. These tools helped quantify each model's predictive prowess and facilitated a nuanced comparison of their strengths and shortcomings.

Overall, these updates signify a rigorous evolution of my methodological approach to mood prediction, reflecting a balance between sophisticated analytical techniques and the practical needs of model deployment and interpretation.

1.11.11 Visualization of Results

Key visualizations include:

- Confusion Matrix: Demonstrated the model's predictive accuracy and areas of misclassification.
- ROC Curves: Illustrated the model's capability to distinguish between the moods across different classification thresholds.

- Precision-Recall Curves: Highlighted the trade-off between precision (the model's ability to avoid mislabeling a sample as positive) and recall (the model's ability to detect all positive instances), especially crucial in the context of our imbalanced dataset.

1.11.12 Key Findings

- **Performance Across Models**: The analysis shows that all three models achieved similar overall accuracy, with SVM slightly outperforming BLR and XGBoost. This suggests that, despite differences in their underlying algorithms, each model can be tuned to reach a competitive level of accuracy in mood prediction from calendar data.
- **Precision and Recall Balance**: For classifying 'Negative' moods (Class 0), XGBoost exhibited the highest precision, indicating its strength in correctly predicting negative moods without as many false positives. However, all models showed similar recall scores for 'Negative' moods, indicating a consistent challenge across models in capturing all true negative instances.
- **Consistency in Predicting 'Positive' Moods**: All models demonstrated equivalent precision and recall for 'Positive' moods (Class 1), with recall scores notably higher than those for 'Negative' moods. This highlights each model's capability to effectively identify 'Positive' instances, underscoring the predictability of factors contributing to positive mood states.
- **Importance of Model Selection and Tuning**: The close performance metrics across models underline the importance of model selection and hyperparameter tuning in achieving optimal results. While SVM marginally led in overall accuracy, the choice between models may also consider aspects like computational efficiency, ease of interpretation, and scalability.
- **Potential for Improvement in Negative Mood Prediction**: The similar recall scores for 'Negative' moods across models suggest a common area for enhancement. Investigating additional features or employing more complex models could potentially improve the identification of negative mood states.
- **Implications for Mood Prediction**: The findings suggest that with careful feature selection and model tuning, machine learning models can effectively predict mood states from calendar data. This opens avenues for applications in personal well-being and productivity optimization by leveraging predictive insights to tailor daily schedules.

1.11.13 3- Executive Summary from the Final Pipeline

Objective The final pipeline aimed to extend the capabilities of the existing framework by incorporating time series analysis into the predictive modeling of mood states based on calendar data. The primary objective was to not only classify mood as positive or negative but also to account for the inherent sequential dependencies present in the data.

Methodology: The methodology involved a structured approach beginning with reloading the calendar data, which was then chronologically sequenced for time series analysis. Preprocessing steps included vectorization of textual data and standardization of numerical features. The dataset was divided into training, validation, and test sets following a 60-20-20 split to ensure a robust assessment of model performance.

Model Selection and Training: Three models were selected for their proficiency in handling time-dependent data:

- Time Series Forest (TSF)

- Long Short-Term Memory networks (LSTM)
- Gated Recurrent Units (GRU)

Each model was trained and tuned. TSF was selected for its interpretability and ensemble learning capabilities; LSTM for its ability to learn long-term dependencies; and GRU for its efficiency and simpler structure.

Performance Evaluation: The performance evaluation was conducted using a variety of metrics:

- Accuracy, precision, recall, and F1-scores were derived from confusion matrices.
- ROC curves and AUC scores gauged the models' ability to distinguish between classes.
- Precision-Recall curves assessed model performance in the context of class imbalance.
- Histograms of predicted probabilities offered insights into model confidence.

Key Insights:

- The TSF model demonstrated superior performance with the highest AUC of 0.77 on the ROC curve, suggesting it was the highest-performing at classification tasks within this context.
- LSTM and GRU models performed similarly to each other but were outperformed by TSF. Their predicted probabilities were less decisive, indicating a potential lack of confidence in classifications.
- However, it is important to consider the difference in model architectures of these models. While LSTM and GRU have RNN-likje structure, TSF doesn't actually predict based on previous sequences. Also, LSTM and GRU were potentially overly complex for this task, highlighing the need to deal with potential overfitting, such as L1/L2 regularization or augmenting the dataset further.

Shortcomings: Despite the advancements, the final pipeline faced several challenges: - The LSTM and GRU models exhibited signs of overfitting, as evidenced by a significant disparity between training and validation accuracy. - The lower AUC values for LSTM (0.49) and GRU (0.47) compared to TSF raised concerns about their ability to generalize effectively.

Conclusion: The final pipeline's exploration into time series analysis enriched the predictive modeling process. The TSF emerged as the standout model due to its high AUC and better performance on precision-recall metrics. However, the shortcomings identified with the LSTM and GRU models underscored the need for further refinement to improve their generalization and predictive confidence. Future work will focus on addressing these limitations, potentially through more extensive hyperparameter tuning, exploring alternative model architectures, or augmenting the dataset to enhance the models' learning and predictive capabilities.

1.11.14 Shortcomings and Future Directions

Shortcomings One of the primary shortcomings encountered in the second pipeline was balancing the dataset's complexity with the depth of insights it could offer. While the binary classification of mood into "positive" and "negative" streamlined the predictive modeling process, it inevitably reduced the granularity of mood analysis. This simplification potentially overlooks the nuanced spectrum of human emotions, where distinct moods like "stressed" vs. "calm" within the same category could have unique triggers and implications. Furthermore, relying on manual labeling for mood categories, despite its personalized accuracy, introduces a subjective bias that might not generalize well across different individuals or contexts. Although a significant step towards capturing the qualitative aspects of events, the incorporation of textual data presented another layer of complexity. Preprocessing such data to extract meaningful patterns while ensuring the models do not get swayed by noise or irrelevant details was a delicate balance that required careful consideration and could still benefit from further refinement.

Another challenge I faced in this pipeline that limited the extension I could make to my first pipeline was in the effort to train an RNN (Recurrent Neural Network) on my calendar entries to try to generate event summaries and descriptions in my own voice. I have spent significant effort on this, but since I tried to include a mixed group of predictors to train the model (numeric data and text data from event descriptions and summaries) and I wanted to generate new entries with, again, a mixed type of data, I ran into lots of complexity issues. Although my training dataset had only ~500 rows, even my Google Colab RAM wasn't enough to fit this model, and it kept crashing. Outside of the technical problems, I also had several concerns about this approach: (1) My data for training this generative model was too limited, and the generated results would be highly overfit. (2) The combined model for handling numeric and text data could become complex, and it could require a lot of fine-tuning and experimentation with architecture and hyperparameters to get high-quality synthetic data.

The shortcomings of the final pipeline can primarily be attributed to the intricate interaction between the complex nature of the machine-learning models employed and the detailed prediction of mood states. The utilization of Time Series Forests (TSF), Long Short-Term Memory networks (LSTMs), and Gated Recurrent Units (GRUs) has indeed enabled advanced temporal pattern recognition. However, this comes at the cost of several substantial limitations that need to be addressed.

A prominent challenge faced with these sophisticated models, especially LSTMs and GRUs, is the risk of overfitting. These models are known for their proficiency in capturing long-term dependencies in sequential data, but this strength also makes them vulnerable to learning the training data too well. This is evidenced by the observed discrepancies between training and validation accuracies, which implies a reduction in the models' ability to generalize to new, unseen data. Furthermore, the computational intensity required for these models cannot be overlooked. Their complex architectures demand considerable computational resources and time, which could be a barrier to efficient training and rapid iteration—a necessity in the fast-paced environment of machine learning and data science.

Another critical issue is the limitations posed by the dataset. With only around 500 rows of data, the training set may not be sufficiently large to effectively train such intricate models, which could lead to a lack of generalization and models that are overfit to the training examples. The pipeline also grapples with the challenges brought about by the integration of numeric and textual data. Maintaining an equilibrium between these different data types, without introducing bias or noise, is a complex task. The integration requires a nuanced approach to feature engineering and model tuning to ensure the models learn from the data without being misled by irrelevant information.

Finally, the approach to mood categorization in the pipeline is rather simplistic, boiling down complex human emotions to a binary classification of positive or negative. This simplification fails to account for the diverse spectrum of human emotions, potentially overlooking the nuanced differences between moods that could be critical in understanding and predicting mood states accurately.

Future Directions For the final pipeline, one avenue to explore is the simplification of the existing model framework. A revised approach might involve less complex models or the development of hybrid methodologies that can effectively capture temporal patterns without the significant computational costs associated with the current models. Such a change would aim to maintain, or even improve, the quality of predictions while streamlining the analytical process.

Advanced regularization techniques, such as dropout layers within the LSTM and GRU models, will be essential to address the overfitting challenge. By incorporating these techniques, I aim to improve the models' ability to generalize by preventing them from relying too heavily on the specific patterns of the training data.

I also recognize the need for a more refined approach to mood classification. Transitioning to a multi-class system for mood categorization could provide a deeper understanding of emotional states and facilitate more accurate predictions, as long as we have rich data. This shift would acknowledge the complexity of human emotions, offering a more precise and granular analysis.

Another focus will be the fine-tuning of the sequential models. I will delve into the nuances of the hyperparameters that manage the models' memory capabilities, adjusting aspects such as the number of layers, hidden units, and learning rates. Such calibration is vital for balancing the retention of relevant information and the prevention of overfitting.

In addition to model tuning, employing k-fold cross-validation techniques will be crucial to ensure that model performance is robust across various data subsets. This validation method will provide a more reliable assessment of the models' predictive power and stability.

Controlling the complexity of models when dealing with mixed data types is another challenge to address. I aim to develop approaches that can separately yet efficiently process numeric and text data within a unified model architecture, enabling each data type to contribute effectively to the overall predictive capability.

Lastly, the application of advanced feature selection and extraction techniques, possibly including dimensionality reduction methods like PCA or the utilization of autoencoders, could reveal new insights by uncovering the latent structures within the data. For the textual data components, more sophisticated Natural Language Processing (NLP) methods, such as leveraging contextual embeddings and sentiment analysis with Universal Sentence Encoder instead of basic TF-IDF, are expected to provide a deeper and more nuanced understanding of the textual information contained in event descriptions.

1.12 Section 12: References

- Buhl, N. (2023, August 22). Time Series Predictions with RNNs. https://encord.com/blog/time-series-predictions-with-recurrent-neural-networks/#:~:text=Recurrent%20Neural%20Networks%20(RNNs)%20offer,dependencies%20and%20tempo
- Chen, T., & Guestrin, C. (2016). XGBoost: A Scalable Tree Boosting System. In Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '16). ACM, New York, NY, USA. https://doi.org/10.1145/2939672.2939785
- DSCM. (n.d.). The Kernel Trick. Retrieved from https://dscm.quora.com/The-Kernel-Trick
- How to download your Google data - Google Account Help. (n.d.). https://support.google.com/accounts/answer/3024190?hl=en

- IBM Developer. (n.d.). https://developer.ibm.com/learningpaths/get-started-time-series-classification-api/what-is-time-series-classification/
- Jurafsky, D., & Martin, J. H. (n.d.). Speech and Language Processing (3rd ed.). Retrieved from https://web.stanford.edu/~jurafsky/slp3/5.pdf
- LeCun, Y., Bengio, Y., & Hinton, G. E. (2015). Deep learning. Nature, 521(7553), 436–444. https://doi.org/10.1038/nature14539
- Multinomial Logistic Regression | STATA Data Analysis Examples. (n.d.). https://stats.oarc.ucla.edu/stata/dae/multinomiallogistic-regression/
- Muralidharan, A. (2019, January 31). Parsing Google Calendar events with Python. Qxf2 BLOG. https://qxf2.com/blog/google-calendar-python/
- Murphy, K. (2012). Machine Learning : A Probabilistic Perspective. http://cds.cern.ch/record/1981503
- Srivatsavaya, P. (2023, July 25). LSTM vs GRU - Prudhviraju Srivatsavaya - Medium. Medium. https://medium.com/@prudhviraju.srivatsavaya/lstm-vs-gru-c1209b8ecb5a#:~:text=LSTM%3A%20LSTM%20has%20more%20parameters,more%20efficient%20for%20la
- Szaitseff. (2019, January 18). Classification of Time Series with LSTM RNN. Kaggle. https://www.kaggle.com/code/szaitseff/classification-of-time-series-with-lstm-rnn Time Series Classification with LSTM Recurrent Neural Networks. (2024, April 2). Omdena. https://www.omdena.com/blog/time-series-classification-model-tutorial

1.13 Datasets

https://drive.google.com/drive/folders/1RPFU1FW_9cd9N-Gq0C1ElE3ytBt3dXeU?usp=sharing

1.14 Use of AI Tools Statement

I utilized ChatGPT-4 extensively in this assignment. Firstly, I used it to generate the code needed for visualizations in this report. Furthermore, I provided key points in bullet form to ChatGPT-4 and asked for help streamlining my written explanations. To aid in building my model, I sought guidance from ChatGPT-4 on library imports for both train/validation/test set splitting, training and testing phases. I also took ChatGPT-4's guidance in distilling academic papers for the mathematical underpinnings of the models I implemented and interpreting the resulting performance metrics. Utilizing ChatGPT-4 was also helpful in generating a concise executive summary, summarizing all my efforts, and adding more to the "Shortcomings and Future Directions" section of my report. Finally, I used Grammarly to proofread and make adjustments.