Social Computing Homework Coursebook

Instructions
Please fill in each exercise and submit the entire document as a PDF on Moodle before the section's respective deadline. Keep working on the whole document so that for the last submission you submit a completely filled in template. You may not change previous sections in subsequent submissions. Some sections require you to work on an existing software project, which you have to fork on [GitHub.com](https://github.com), or clone and create your repository. Provide the URL of your public fork or repository of this project below.

Fill in each answer to a homework task to the textbox underneath. Use as much space as you wish. Do not provide long code snippets or other irrelevant information.

Restrictions
You may use AI tools for language styling or only. Usage of any AI tools to answer questions, inspire creative solutions or write code is strictly forbidden. Group work and sharing solutions is strictly prohibited. Any suspected cases of [misconduct]() will be referred to the Education Dean. If you are not sure whether you are in violation of course-specific restrictions or the university's code of conduct, please ask the Lecturer or a TA.

Name

Vu Truong

Student ID

2506393

Student Email

Vu.Truong@student.oulu.fi

GitHub Repository URL

https://github.com/HarryxDD/social-computing-hw
https://github.com/HarryxDD/mini_social_exercise

AI Use Disclaimer

Explain in detail in what parts and how AI was used for any of the work above. Fill it out and update after each homework submission, even if you did not use AI at all.

Your answers to homework tasks should not include AI-generated code or text.

I used Social Computing AI Agent to ask about the task 1.1 because there are no "purpose" description inside the database.

For task 2.1, I used AI Agent to ask about the growing trend of some social media platforms.

For task 4.1, after my LDA model generated the keyword lists for each topic, I used an AI Agent to suggest potential human-readable titles

Task 1 (due 22.9.2025 23:59) 15 points

Exercise 1.1 Reading the dataset: Load the database and for each table, print and inspect the available columns and the number of rows. Explain below how you loaded the database. For each table, describe all columns (name, purpose, type, example of contents). You may use SQL and/or Python to perform this task. (3 points)

```
# Explanations for the work are being added as comments
import sqlite3
import pandas as pd

# Current db file location (same location as the code file)
dbfile = 'database.sqlite'
# Establish a connection to the db
conn = sqlite3.connect(dbfile)

# Read all table names -> turn it to a dataframe
tablenames_df = pd.read_sql_query("SELECT name FROM sqlite_master WHERE
type='table';", conn)

# Convert df to a list
tables = tablenames_df['name'].tolist()

for table in tables:
    print(f"Table: {table}")
    df = pd.read_sql_query(f"SELECT * FROM {table};", conn)
    # Inspect the table
    print(f"Number of rows: {len(df)}")
    print(f"Available columns: {df.columns.tolist()}")
    # Get metadata
    col = pd.read_sql_query(f"PRAGMA table_info({table});", conn)
    for idx, row in col.iterrows():
        print(f"Name: {row['name']}")
        print(f"Type: {row['type']}")
        # Hardcoded purpose as metadata is not available in db so I will
describe this in the output
        print(f"Purpose: -")
        print(f"Example: {df[row['name']].head(1).values[0]}")
        print("--")
    print("-----")

"""
Output:
```

```
Table: follows
Note: This is a many-to-many relationship table between users and their
followers
Number of rows: 7225
Available columns: ['follower_id', 'followed_id']
Name: follower_id
Type: INT
Purpose: This is the id of the user who is following
Example: 12
--
Name: followed_id
Type: INT
Purpose: This is the id of the user who is being followed
Example: 1
--
-----
Table: users
Number of rows: 210
Available columns: ['id', 'username', 'location', 'birthdate', 'created_at',
'profile', 'password']
Name: id
Type: INT
Purpose: Id of the user
Example: 1
--
Name: username
Type: varchar(50)
Purpose: Username of user
Example: artistic_amy
--
Name: location
Type: varchar(100)
Purpose: Location of user
Example: Boston, USA
--
Name: birthdate
Type: date
Purpose: User's date of birth
Example: 1997-06-30
--
Name: created_at
Type: timestamp
Purpose: The timestamp when the user account was created
```

```
Example: 2022-07-01 12:17:48
--
Name: profile
Type: TEXT
Purpose: Profile description of user that contains personality traits and
interests
Example: Artistic soul from Boston ? | Born in '97 | Balancing mind & style |
Fashion lover | News junkie | Embracing the highs and lows | Dreaming big,
moving forward ✨
--
Name: password
Type: TEXT
Purpose: Password for the account
Example: izmQoLHw
--
-----
Table: sqlite_sequence
Note: Automatically created table manage AUTOINCREMENT fields
Number of rows: 3
Available columns: ['name', 'seq']
Name: name
Type:
Purpose: Shows which table (like reactions, posts, ect) the row is about
Example: reactions
--
Name: seq
Type:
Purpose: Shows the last used AUTOINCREMENT value for that table
Example: 8286
--
-----
Table: reactions
Number of rows: 8276
Available columns: ['id', 'post_id', 'user_id', 'reaction_type']
Name: id
Type: INTEGER
Purpose: Id of the reaction
Example: 1
--
Name: post_id
Type: INTEGER
Purpose: Id of the post that the reaction is for
Example: 2631
--
```

```
Name: user_id
Type: INTEGER
Purpose: Id of the user who made the reaction
Example: 60
--
Name: reaction_type
Type: TEXT
Purpose: The type of reaction
Example: like
--
-----
Table: comments
Number of rows: 5804
Available columns: ['id', 'post_id', 'user_id', 'content', 'created_at']
Name: id
Type: INTEGER
Purpose: Id of the comment
Example: 1
--
Name: post_id
Type: INTEGER
Purpose: Id of the post that the comment is for
Example: 1963
--
Name: user_id
Type: INTEGER
Purpose: Id of the user who commented
Example: 55
--
Name: content
Type: TEXT
Purpose: Content of the comment
Example: Haha, I bet your neighbors are either loving or hating you right now!
Crank it up and see if you can get a dance party going next door. #DIYparty
--
Name: created_at
Type: TIMESTAMP
Purpose: The timestamp when the comment was created
Example: 2022-12-04 02:36:15
--
-----
Table: posts
Number of rows: 1303
Available columns: ['id', 'user_id', 'content', 'created_at']
```

```
Name: id
Type: INTEGER
Purpose: Id of the post
Example: 1718
--
Name: user_id
Type: INTEGER
Purpose: Id of the post owner
Example: 10
--
Name: content
Type: TEXT
Purpose: Content of the post
Example: Just had the most ridiculous encounter with a cat in Shibuya. It
hissed like I was invading its turf! #CatWhisperer #TokyoLife
--
Name: created_at
Type: TIMESTAMP
Purpose: The timestamp when the post was created
Example: 2023-10-12 10:43:24
--
-----

"""
```

Exercise 1.2 Lurkers: How many users are there on the platform who have not interacted with posts or posted any content yet (but may have followed other users)? Answer and explain your queries/calculations below. You may use SQL and/or Python to perform this task. (3 points)

```
# Explanations for the work are being added as comments
import sqlite3
import pandas as pd

# Current db file location
dbfile = 'database.sqlite'
# Establish a connection to the db
conn = sqlite3.connect(dbfile)

try:
    # Check for users who not exist in posts, comments, and reactions table
using subqueries
```

```
    lurkers = pd.read_sql_query("""
    SELECT
        id
    FROM users
    WHERE id NOT IN (SELECT user_id FROM posts)
    AND id NOT IN (SELECT user_id FROM comments)
    AND id NOT IN (SELECT user_id FROM reactions);
    """, conn)
    # print("Lurkers: ")
    # print(lurkers)
    print("The number of people who have not interacted at all: ",
len(lurkers))
except Exception as e:
    print(f"Error: {e}")

"""
Output:
The number of people who have not interacted at all:  55

"""
```

Exercise 1.3 Influencers: In the history of the platform, who are the 5 users with the most engagement on their posts? Describe how you measure engagement. Answer and explain your queries/calculations below. You may use SQL and/or Python to perform this task. (4 points)

```
# Explanations for the work are being added as comments
import sqlite3
import pandas as pd

# Current db file location
dbfile = 'database.sqlite'
# Establish a connection to the db
conn = sqlite3.connect(dbfile)

"""
To find top 5 influencers, I count the number of reactions and comments on each
user's posts.
First, I JOIN the posts table with the users table to get the username (the
author).
Then, I LEFT JOIN the reactions and comments tables to count the number of
reactions and comments for each posts.
```

```
Finally, I group the results by username and order them by the total number of
reactions and comments in descending order, limiting the results to the top 5.

By using DISTINCT in the COUNT, I ensure that each reaction and is counted only
once, because when joining multiple tables, there can be duplicate rows for the
same reaction and comment, resulting in same count value for these columns.
"""

try:
    influencer_df = pd.read_sql_query("""
    SELECT
        users.id,
        users.username,
        COUNT(DISTINCT reactions.id) as Reactions,
        COUNT(DISTINCT comments.id) AS Comments
    FROM posts
    JOIN users on users.id = posts.user_id
    LEFT JOIN reactions on posts.id = reactions.post_id
    LEFT JOIN comments ON posts.id = comments.post_id
    GROUP by users.username
    ORDER BY (COUNT(DISTINCT reactions.id) + COUNT(DISTINCT comments.id)) DESC
    LIMIT 5;
    """, conn)
    print("Top 5 influencers: ")
    print(influencer_df)
except Exception as e:
    print(f"Error: {e}")

"""
Output:
Top 5 influencers:
   id     username  Reactions  Comments
0  54    WinterWolf        267       179
1  65   PinkPanther        234       152
2  94     PinkPetal        246       137
3  81  GoldenDreams        217       149
4  30     WildHorse        196       157

"""
```

Exercise 1.4 Spammers: Identify users who have shared the same text in posts or comments at least 3 times over and over again (in all their history, not just the last 3 contributions). Answer and explain your queries/calculations below. You may use SQL and/or Python to perform this task. (5 points)

```python
# Explanations for the work are being added as comments
import sqlite3
import pandas as pd

# Current db file location
dbfile = 'database.sqlite'
# Establish a connection to the db
conn = sqlite3.connect(dbfile)

"""
For this task, I identify spammer by check the same contents being posted or
commented more than 3 times by the same user
I use 2 separate SELECT to find the spam and combine them using UNION.
I also add a column 'type' to indicate whether the spam is from post or
comment.
"""

try:
    spammer_df = pd.read_sql_query("""
    SELECT
        users.username,
        posts.content,
        'post' as type,
        COUNT(*) as occur
    FROM posts
    JOIN users on users.id = posts.user_id
    GROUP by posts.user_id, posts.content
    HAVING COUNT(*) >= 3

    UNION

    SELECT
        users.username,
        comments.content,
        'comment' as type,
        COUNT(*) as occur
    FROM comments
    JOIN users on users.id = comments.user_id
    GROUP by comments.user_id, comments.content
    HAVING COUNT(*) >= 3;
    """, conn)
    print("Spammer: ")
    print(spammer_df)
```

```
except Exception as e:
    print(f"Error: {e}")

"""
Output:
Spammer:
        username                                                    content     type  o
ccur
0    coding_whiz  ?FREE VACATION? Tag a friend you’d take to
Bal...   comment      3
1    coding_whiz  Shocking! #lol #weekend #coffee #bookstagram
#...     post      3
2    coding_whiz  Top 10 gadgets of 2025 – All available here:
b...     post      8
3    eco_warrior  Not gonna lie, I was skeptical at first. But
a...     post      6
4    eco_warrior  Revolutionary idea! #fashionblogger
#instafash...     post      3
5    eco_warrior  Wearing this hoodie in my latest reel—so many
...     post      4
6   history_buff  A lot of you asked what helped me drop 5kg in
...     post      5
7   history_buff  Best way to clean your sneakers ? snag yours
h...     post      5
8   history_buff  Mood: me refreshing for likes every 30
seconds...     post      5
9   history_buff  What do you think? #thoughts
#motivationmonday...     post      4
10  history_buff  You need this travel pillow in your life ?
sho...     post      3
11     night_owl  ? Mega Giveaway Alert! ? Follow all accounts
w...     post      8
12     night_owl  ?FLASH GIVEAWAY? Click the link in our bio to
...     post      5
13     night_owl  Find out why everyone is switching to this
new...     post      4
14     night_owl  This one trick will make you $500/day from
hom...     post      3
15     yoga_yogi  I couldn’t believe it! I just entered this
giv...     post      5
16     yoga_yogi  Just entered this Xbox giveaway and the form
w...     post      3

"""
```

Task 2 (due 29.9.2025 23:59) 15 points

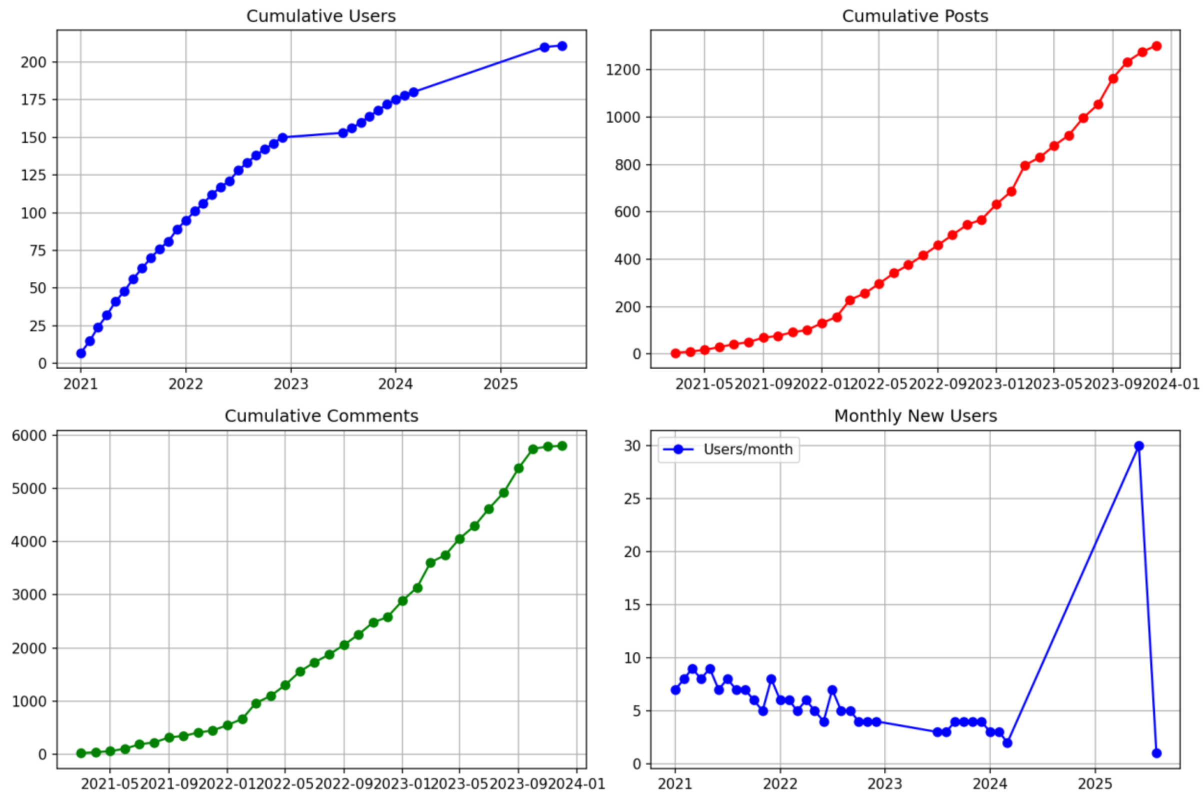
Exercise 2.1 Growth: This year, we are renting 16 servers to run our social media platform. They are soon at 100% capacity, so we need to rent more servers. We would like to rent enough to last for 3 more years without upgrades, plus 20% capacity for redundancy. We need an estimate of how many servers we need to start renting based on past growth trends. Plot the trend on a graph using Python and include it below. Answer and explain your queries/calculations below. You may use SQL and/or Python to perform this task. (Note that the dataset may not end in the current year, please assume that the last data marks today's date) (3 points)

```python
import sqlite3
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np

"""
For this task, I thought about the growing factors, is it linear growth or exponential growth, as normally some social media platforms grow exponentially in the beginning, but after a while, the growth rate slows down.
After analyzing the data, I found that the growth is more linear than exponential. So I decided to use a linear projection for the next 3 years.

And the answer for the number of additional servers needed is 23. The calculation will be shown below.

"""


def get_data():
    conn = sqlite3.connect('database.sqlite')

    # This query get total counts of users, posts, and comments.
    totals = pd.read_sql_query("SELECT (SELECT COUNT(*) FROM users) as users, (SELECT COUNT(*) FROM posts) as posts, (SELECT COUNT(*) FROM comments) as comments", conn)

    # These queries get monthly new users, posts, and comments.
```

```
    monthly_users = pd.read_sql_query("SELECT strftime('%Y-%m', created_at) as
month, COUNT(*) as count FROM users GROUP BY strftime('%Y-%m', created_at)
ORDER BY month", conn)
    monthly_posts = pd.read_sql_query("SELECT strftime('%Y-%m', created_at) as
month, COUNT(*) as count FROM posts GROUP BY strftime('%Y-%m', created_at)
ORDER BY month", conn)
    monthly_comments = pd.read_sql_query("SELECT strftime('%Y-%m', created_at)
as month, COUNT(*) as count FROM comments GROUP BY strftime('%Y-%m',
created_at) ORDER BY month", conn)

    conn.close()

    return totals.iloc[0]['users'], totals.iloc[0]['posts'],
totals.iloc[0]['comments'], monthly_users, monthly_posts, monthly_comments

def calculate_projections(total_users, total_posts, total_comments,
monthly_users):
    # The value 1.0 is based on the assumption that each user has many props,
such as posts, comments, authentication, etc.
    user_weight = 1.0

    # For posts, each of them can contains long text, images, and interactions.
    post_weight = 0.5

    # For comments, they are usually short text, but can also contain images,
and reactions.
    comment_weight = 0.2

    # Traffic spike factor to account for peak times when user activity is
higher.
    traffic_spike_factor = 1.2

    # Current server load
    current_load = (total_users * user_weight + total_posts * post_weight +
total_comments * comment_weight) * traffic_spike_factor

    # Continue current growth for 3 years
    days_until_now = len(monthly_users) * 30
    daily_user_growth = total_users / days_until_now

    # Projected number of users for the next 3 years
    projected_users = total_users + (daily_user_growth * 1095)

    # Projected posts and comments based on user growth
```

```python
    user_growth_multiplier = projected_users / total_users
    projected_posts = total_posts * user_growth_multiplier
    projected_comments = total_comments * user_growth_multiplier

    # Calculate projected server load and servers needed
    projected_load = (projected_users * user_weight + projected_posts *
post_weight + projected_comments * comment_weight) * traffic_spike_factor

    # Current servers with 20% redundancy
    needed_servers = 16 * (projected_load / current_load) * 1.2

    return {
        'users': projected_users,
        'posts': projected_posts,
        'comments': projected_comments,
        'needed_servers': needed_servers
    }

def create_plots(monthly_users, monthly_posts, monthly_comments):
    fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2, figsize=(12, 8))

    for df in [monthly_users, monthly_posts, monthly_comments]:
        df['date'] = pd.to_datetime(df['month'])
        df['cumulative'] = df['count'].cumsum()

    # Create 4 plots
    ax1.plot(monthly_users['date'], monthly_users['cumulative'], 'b-o')
    ax1.set_title('Cumulative Users'); ax1.grid(True)

    ax2.plot(monthly_posts['date'], monthly_posts['cumulative'], 'r-o')
    ax2.set_title('Cumulative Posts'); ax2.grid(True)

    ax3.plot(monthly_comments['date'], monthly_comments['cumulative'], 'g-o')
    ax3.set_title('Cumulative Comments'); ax3.grid(True)

    ax4.plot(monthly_users['date'], monthly_users['count'], 'b-o',
label='Users/month')
    ax4.set_title('Monthly New Users'); ax4.grid(True); ax4.legend()

    plt.tight_layout()
    plt.savefig('growth_analysis.png', dpi=150)
    plt.show()

def analyze_and_plot():
```

```python
    total_users, total_posts, total_comments, monthly_users, monthly_posts,
monthly_comments = get_data()

    print(f"Current: {total_users} users, {total_posts} posts, {total_comments}
comments")

    results = calculate_projections(total_users, total_posts, total_comments,
monthly_users)

    print(f"\n3-Year Linear Projection:")
    print(f"  Users: {results['users']:.0f}, Posts: {results['posts']:.0f},
Comments: {results['comments']:.0f}")
    print(f"  Additional servers needed: +{results['needed_servers'] -
16:.0f}")
    print(f"  Total servers: {results['needed_servers']:.0f}")

    create_plots(monthly_users, monthly_posts, monthly_comments)

if __name__ == "__main__":
    analyze_and_plot()

"""
Output:
Current: 211 users, 1303 posts, 5804 comments

3-Year Linear Projection:
  Users: 431, Posts: 2662, Comments: 11857
  Additional servers needed: +23
  Total servers: 39

"""
```



Exercise 2.2 Virality: Identify the 3 most viral posts in the history of the platform. Select and justify a specific metric or requirements for a post to be considered viral. Answer and explain your queries/calculations below. You may use SQL and/or Python to perform this task. (4 points)

```python
# Explainations for the work are being added as comments
import sqlite3
import pandas as pd

# Current db file location
dbfile = 'database.sqlite'
# Establish a connection to the db
conn = sqlite3.connect(dbfile)

"""
After research about viral post, I found that it is a piece of content that
gets shared quickly across various social media platforms in a short period of
time. So I decided to use growth rate in the first few hours to measure the
virality of a post.
```

```
I was trying to calculate the growth rate based on reactions, but I found that
the table does not have a created_at column, so I can only use comments in this
case.
"""

CALCULATING_HOURS = 24

def calculate_growth_rate_hours(table_alias, post_alias, hours):
    # Don't forget to check if the hours since posted is less than the
calculating hours
    return f"""
    COUNT(DISTINCT CASE WHEN (julianday({table_alias}.created_at) -
julianday({post_alias}.created_at)) * 24 <= {hours} THEN {table_alias}.id END)
* 1.0 /
    CASE
        WHEN (julianday('now') - julianday({post_alias}.created_at)) * 24 >=
{hours} THEN {hours}
        WHEN (julianday('now') - julianday({post_alias}.created_at)) * 24 < 1
THEN 1
        ELSE (julianday('now') - julianday({post_alias}.created_at)) * 24
    END
    """

try:
    viral_post_df = pd.read_sql_query(f"""
    SELECT
        p.id,

        -- Total engagement (comments + reactions)
        COUNT(DISTINCT c.id) as total_comments,
        COUNT(DISTINCT r.id) as total_reactions,
        (COUNT(DISTINCT c.id) + COUNT(DISTINCT r.id)) as absolute_engagement,

        -- Growth rate: comments per hour in first {CALCULATING_HOURS} hours
        {calculate_growth_rate_hours('c', 'p', CALCULATING_HOURS)} as
growth_rate,

        -- Combined virality score
        {calculate_growth_rate_hours('c', 'p', CALCULATING_HOURS)} *
(COUNT(DISTINCT c.id) + COUNT(DISTINCT r.id)) as virality_score
    FROM posts p
    LEFT JOIN comments c on c.post_id = p.id
    LEFT JOIN reactions r on r.post_id = p.id
    GROUP by p.id
```

```
    HAVING absolute_engagement > 0
    ORDER BY virality_score DESC
    LIMIT 3;
    """, conn)
    print(f"Viral posts - first {CALCULATING_HOURS} hours: ")
    print(viral_post_df)
except Exception as e:
    print(f"Error: {e}")

"""
Output:

Viral posts - first 5 hours:
     id  total_comments  total_reactions  absolute_engagement  growth_rate  vir
ality_score
0  2351              62              139                  201         12.4
     2492.4
1  2813              82              103                  185         12.0
     2220.0
2  2195              45              133                  178          9.0
     1602.0

Viral posts - first 12 hours:
     id  total_comments  total_reactions  absolute_engagement  growth_rate  vir
ality_score
0  2813              82              103                  185     6.833333
1264.166667
1  2351              62              139                  201     5.166667
1038.500000
2  2004              71               94                  165     5.916667
 976.250000

Viral posts - first 24 hours:
     id  total_comments  total_reactions  absolute_engagement  growth_rate  vir
ality_score
0  2813              82              103                  185     3.416667
 632.083333
1  2351              62              139                  201     2.583333
 519.250000
2  2004              71               94                  165     2.958333
 488.125000

As we can see, the vital posts are consistent across different hours, so the
answer for the question is post id 2813, 2351, and 2004. There was a slight
```

```
change in the order because there's a higher early burst of post id 2351 at the
start, but slower sustained growth.
"""
```

Exercise 2.3 Content Lifecycle: What is the average time between the publishing of a post and the first engagement it receives? What is the average time between the publishing of a post and the last engagement it receives? Answer and explain your queries/calculations below. You may use SQL and/or Python to perform this task. (4 points)

```
# Explainations for the work are being added as comments
import sqlite3
import pandas as pd

# Current db file location
dbfile = 'database.sqlite'
# Establish a connection to the db
conn = sqlite3.connect(dbfile)

"""
For this task, I define the engagement based on comments since the reactions
table does not have a created_at column.

I excluded posts that have no comments, since they do not have any engagement,
but still show the number of such posts in the output.

Basically, I created a CTE to calculate the time to first comment and time to
last comment for each post, then I used aggregate functions to get the required
metrics.

I used INNER JOIN to exclude posts with no comments first, then I calculated
the number of such posts by subtracting from the total.
"""

try:
    content_lifecycle = pd.read_sql_query(f"""
    with post_lifecycle as (
    SELECT
        p.id,
        p.created_at,
        MIN(c.created_at) AS first_comment_at,
```

```
        (julianday(MIN(c.created_at)) - julianday(p.created_at)) * 24 as
hours_to_first_comment,
        MAX(c.created_at) as last_comment_at,
        (julianday(MAX(c.created_at)) - julianday(p.created_at)) * 24 as
hours_to_last_comment
    from posts p
    INNER join comments c on p.id = c.post_id
    GROUP by p.id
    )
    SELECT
        COUNT(*) as posts_with_comments,
        (select COUNT(*) from posts) - count(*) as posts_with_no_comments,
        AVG(hours_to_first_comment) as avg_hr_to_first_cmt,
        AVG(hours_to_last_comment) as avg_hr_to_last_cmt
    from post_lifecycle;
    """, conn)
    print(f"Content Lifecycle: ")
    print(content_lifecycle)
except Exception as e:
    print(f"Error: {e}")

"""
Output:
Content Lifecycle:
   posts_with_comments  posts_with_no_comments  avg_hr_to_first_cmt  avg_hr_to_
last_cmt
0                 1215                      88            86.604362          15
1.445664

"""
```

Exercise 2.4 Connections: Identify the top 3 user pairs who engage with each other's content the most. Define and describe your metric for engagement. Answer and explain your queries/calculations below. You may use SQL and/or Python to perform this task. (4 points)

```
# Explainations for the work are being added as comments
import sqlite3
import pandas as pd

# Current db file location
dbfile = 'database.sqlite'
# Establish a connection to the db
```

```python
conn = sqlite3.connect(dbfile)

"""
For this task, I define engagement as the total number of comments and
reactions exchanged between two users on each other's posts. This means I count
all individual comments and reactions that flow in both directions between a
user pair.

First, I create the CTE all_engagements to gather all comments and reactions
between users, ensuring that self-engagements are excluded by using WHERE
(c|r).user_id != p.user_id.

The second CTE user_pairs aggregates the total engagement between each pair of
users.
For example, if User A commented 2 times and reacted 3 times to User B's posts,
the total engagement from User A to User B would be 5.

The third CTE mutual_engagement combines the engagements from both users in
each pair to get the total mutual engagement. I joined the user_pairs table
with itself to achieve this. I avoid double counting by ensuring that I only
consider pairs where action_owner < post_owner (or action_owner > post_owner no
matter), so each user pair appears only once in the final results regardless of
who initiated more engagement.

"""

try:
    connections = pd.read_sql_query(f"""
    WITH all_engagements AS (
    SELECT
        c.user_id AS action_owner,
        p.user_id AS post_owner,
        'comment' AS type,
        count(*) AS quantity
    FROM comments c
    JOIN posts p ON p.id = c.post_id
    WHERE c.user_id != p.user_id
    GROUP BY c.user_id, p.user_id

    UNION ALL

    SELECT
        r.user_id AS action_owner,
        p.user_id AS post_owner,
```

```
        'reaction' AS type,
        count(*) AS quantity
    FROM reactions r
    JOIN posts p ON p.id = r.post_id
    WHERE r.user_id != p.user_id
    GROUP BY r.user_id, p.user_id
    ),
    user_pairs as (
    SELECT
        action_owner,
        post_owner,
        SUM(quantity) AS total_engagement
    FROM all_engagements
    GROUP BY action_owner, post_owner
    ),
    mutual_engagement AS (
    SELECT
        CASE WHEN e1.action_owner < e1.post_owner THEN e1.action_owner ELSE
e1.post_owner END AS user1_id,
        CASE WHEN e1.action_owner < e1.post_owner THEN e1.post_owner ELSE
e1.action_owner END AS user2_id,
        e1.total_engagement + e2.total_engagement AS mutual_total
    FROM user_pairs e1
    JOIN user_pairs e2 ON e1.action_owner = e2.post_owner AND e1.post_owner =
e2.action_owner
    WHERE e1.action_owner < e1.post_owner
    )

    SELECT
        u1.username AS user1,
        u2.username AS user2,
        me.mutual_total AS total_mutual_engagement
    FROM mutual_engagement me
    JOIN users u1 ON me.user1_id = u1.id
    JOIN users u2 ON me.user2_id = u2.id
    ORDER BY me.mutual_total DESC
    LIMIT 3;
    """, conn)
    print(f"Connections: ")
    print(connections)
except Exception as e:
    print(f"Error: {e}")

"""
```

```
Output:
Connections:
           user1       user2  total_mutual_engagement
0  DancingDolphin  SilverMoon                       16
1     userInBlack    TigerEye                       13
2       StarGazer  WinterWolf                       13

"""
```

Task 3 (due 19.10.2025 23:59) 15 points

Exercise 3.1 Censorship: implement the `moderate_content` function that automatically detects and censors inappropriate user posts on the platform. Your function should take a post, comment or user introduction as input and apply censorship rules to either clean or remove content, and supply a risk score that corresponds to the number and weight of violations in the content (note the risk classification thresholds in the code). The exact rules are detailed on the Rules page. Think of and implement one more moderation measure you think is important to keep the platform safe. Include and explain your implementation below. (5 points)

```python
def moderate_content(content):
    """
    Args
        content: the text content of a post or comment to be moderated.

    Returns:
        A tuple containing the moderated content (string) and a severity score
(float). There are no strict rules or bounds to the severity score, other than
that a score of less than 1.0 means no risk, 1.0 to 3.0 is low risk, 3.0 to 5.0
is medium risk and above 5.0 is high risk.

    This function moderates a string of content and calculates a severity score
based on
    rules loaded from the 'censorship.dat' file. These are already loaded as
TIER1_WORDS, TIER2_PHRASES and TIER3_WORDS. Tier 1 corresponds to strong
profanity, Tier 2 to scam/spam phrases and Tier 3 to mild profanity.

    You will be able to check the scores by logging in with the administrator
account:
            username: admin
            password: admin
    Then, navigate to the /admin endpoint. (http://localhost:8080/admin)
    """

    # Handle empty or invalid content
    if not content or not isinstance(content, str):
        return content, 0.0

    moderated_content = content
    score = 0.0
```

```
    """
    Rule 1.1.1
    A case-insensitive, whole-word search is performed against the Tier 1 Word
List. If a match is found, the function immediately returns the string [content
removed due to severe violation] and a fixed Content Score of 5.0.
    """
    for word in TIER1_WORDS:
        pattern = r'\b' + re.escape(word) + r'\b'
        if re.search(pattern, content, re.IGNORECASE):
            return "[content removed due to severe violation]", 5.0

    """
    Rule 1.1.2
    If no Tier 1 match is found, a case-insensitive, whole-phrase search is
performed against the Tier 2 Phrase List. If a match is found, the function
immediately returns the string [content removed due to spam/scam policy] and a
fixed Content Score of 5.0.
    """
    for phrase in TIER2_PHRASES:
        # Use word boundaries for whole phrase matching
        pattern = r'\b' + re.escape(phrase) + r'\b'
        if re.search(pattern, content, re.IGNORECASE):
            return "[content removed due to spam/scam policy]", 5.0

    """
    Rule 1.2.1
    Each case-insensitive, whole-word match from the Tier 3 Word List is
replaced with asterisks (*) equal to its length. The Content Score is
incremented by +2.0 for each match.
    """
    for word in TIER3_WORDS:
        pattern = r'\b' + re.escape(word) + r'\b'
        matches = re.findall(pattern, moderated_content, re.IGNORECASE)
        if matches:
            score += len(matches) * 2.0
            def replace_with_asterisks(match):
                return '*' * len(match.group(0))
            moderated_content = re.sub(pattern, replace_with_asterisks,
moderated_content, flags=re.IGNORECASE)

    """
    Rule 1.2.2
```

```
    Each detected URL is replaced with [link removed]. The Content Score is
incremented by +2.0 for each match.

    After detecting some odd URLs, I decided to implement some enhanced URL
detection that checks for:
    - Full URLs with and without http(s) protocol
    - Obfuscated URLs: example[.]com, domain[dot]org (spammer technique to
bypass filters)
    - Common TLDs: .com, .org, .net, .edu, .gov, .io, .co.uk, .co.jp, etc.
    - Excludes URLs inside square brackets [example.com] to avoid false
positives


    # Here I de-obfuscate URLs by replacing [.] and [dot] with actual dots
    # I temporarily convert these to domain.com so our pattern can detect them
    """
    deobfuscated_content = moderated_content
    deobfuscated_content = re.sub(r'\[\.\]', '.', deobfuscated_content)
    deobfuscated_content = re.sub(r'\[dot\]', '.', deobfuscated_content,
flags=re.IGNORECASE)

    """
    Regex pattern explaination:
    https?://[^\s\[\]]+  -> Matches full URLs starting with http:// or https://
    www\.[a-zA-Z0-9][-a-zA-Z0-9.]*[a-zA-Z0-9] -> Matches URLs starting with
www.
    \b[a-zA-Z0-9][-a-zA-Z0-9]*\.[a-z]{2,}(?:\.[a-z]{2,})? -> Matches domain.abc
or domain.abc.abc
    """
    url_pattern = r'(?<![@\[])(?:https?://[^\s\[\]]+|www\.[a-zA-Z0-9][-a-zA-Z0-
9.]*[a-zA-Z0-9](?:/[^\s\[\]]*)?|\b[a-zA-Z0-9][-a-zA-Z0-9]*\.[a-z]{2,}(?:\.[a-
z]{2,})?(?:/[^\s\[\]]*)?)(?!\])'

    urls = re.findall(url_pattern, deobfuscated_content, re.IGNORECASE)
    if urls:
        url_count = len(urls)
        score += url_count * 2.0
        moderated_content = re.sub(url_pattern, '[link removed]',
deobfuscated_content, flags=re.IGNORECASE)

    """
    Rule 1.2.3
```

```
    If content has >15 alphabetic characters and >70% are uppercase, the
Content Score is incremented by a fixed value of +0.5. The content is not
modified.
    """
    alphabetic_chars = [c for c in moderated_content if c.isalpha()]
    if len(alphabetic_chars) > 15:
        uppercase_count = sum(1 for c in alphabetic_chars if c.isupper())
        uppercase_ratio = uppercase_count / len(alphabetic_chars)
        if uppercase_ratio > 0.7:
            score += 0.5

    """
    Additional measure: Giveaway/Contest Spam Detection
    After investigating the dataset, I found that giveaway and contest spam is
a probable issue on this platform, because it can lead to harmful outcomes for
users. To name a few: leading to phising attempts, create false expectations
and disappointment, etc.

    Real examples from the platform that currently score 0.0 but are clearly
spam:
    - "FLASH GIVEAWAY? Click the link in our bio to claim your PS5! Only 100
units left!"
    - "We're giving away $1000 to 5 lucky people! Like, share, and comment
'WIN' to enter!"

    Penalty: +2.0 (severe spam that harms user trust and security)
    """

    # Define giveaway spam patterns with their regex
    giveaway_patterns = [
        r'\bgiveaway\b',
        r'\bgiving away\b',
        r'\bwin free\b',
        r'\bclaim your\b',
        r'\bclick\s+(the\s+)?link\b',
        r'\b(dm|message)\s+(us|me)\b',
        r'\bfollow\s+and\b',
        r'\benter\s+to\s+win\b',
        r'\bonly\s+\d+\s+(left|units)\b',
        r'\bflash\s+giveaway\b',
        r'\bcontest\s+alert\b',
        r'\blucky\s+(winner|people)\b',
    ]
```

```
    giveaway_matches = 0
    content_lower = content.lower()

    """
    I count the number of giveaway-related patterns matched in the content. If
2 or more patterns are found, I consider it as giveaway spam and increment the
score by +2.0
    """
    for pattern in giveaway_patterns:
        if re.search(pattern, content_lower):
            giveaway_matches += 1

    if giveaway_matches >= 2:
        score += 2.0

    return moderated_content, score
```

Exercise 3.2 User risk analysis: Assign risk scores to each user by implementing the `user_risk_analysis` function. This function returns a risk score for a given user based on rules presented on the Rules page. Identify the top 5 highest risk users. Think of and implement one more risk prediction measure you think is important to keep the platform safe. Answer and explain your queries/calculations below. (5 points)

```
def user_risk_analysis(user_id):
    """
    Args:
        user_id: The ID of the user on which we perform risk analysis.

    Returns:
        A float number score showing the risk associated with this user. There
are no strict rules or bounds to this score, other than that a score of less
than 1.0 means no risk, 1.0 to 3.0 is low risk, 3.0 to 5.0 is medium risk and
above 5.0 is high risk. (An upper bound of 5.0 is applied to this score
elsewhere in the codebase)

        You will be able to check the scores by logging in with the
administrator account:
            username: admin
            password: admin
        Then, navigate to the /admin endpoint. (http://localhost:8080/admin)
    """
```

```
    user = query_db('SELECT profile, created_at FROM users WHERE id = ?',
(user_id,), one=True)
    if not user:
        return 0.0

    # Step 1: I moderate the user's profile description and get the score from
it
    profile_text = user['profile'] if user['profile'] else ''
    _, profile_score = moderate_content(profile_text)

    # Step 2: I moderate all posts made by the user and calculate the average
post score by iterating through each post, moderating its content, and
collecting the scores to compute the average
    posts = query_db('SELECT content FROM posts WHERE user_id = ?', (user_id,))
    if posts and len(posts) > 0:
        post_scores = []
        for post in posts:
            _, post_score = moderate_content(post['content'])
            post_scores.append(post_score)
        average_post_score = sum(post_scores) / len(post_scores)
    else:
        average_post_score = 0.0

    # Step 3: I moderate all comments and get the average comment score just
like posts
    comments = query_db('SELECT content FROM comments WHERE user_id = ?',
(user_id,))
    if comments and len(comments) > 0:
        comment_scores = []
        for comment in comments:
            _, comment_score = moderate_content(comment['content'])
            comment_scores.append(comment_score)
        average_comment_score = sum(comment_scores) / len(comment_scores)
    else:
        average_comment_score = 0.0

    # Step 4: I calculate the content risk score using weighted contributions
from profile, posts, and comments
    content_risk_score = (profile_score * 1) + (average_post_score * 3) +
(average_comment_score * 1)

    # Step 5: I adjust the risk score based on account age
    user_created_at = user['created_at']
```

```
    account_age_days = (datetime.utcnow() - user_created_at).days

    if account_age_days < 7:
        user_risk_score = content_risk_score * 1.5
    elif account_age_days < 30:
        user_risk_score = content_risk_score * 1.2
    else:
        user_risk_score = content_risk_score

    """
    Additional Risk Measure
    This detects automated spam bots that post at unnaturally high frequencies.
    I decided to implement this based on research into bot behavior patterns.
    There are some reason that this might affect negatively to the platform:
    - Bots can post clean content that evades keyword filters
    - Make the platform less appealing to real users
    - Bots can flood the platform with spam even if content seems clean
    """

    suspicious_activity_score = 0.0

    """
    First, I check the posting frequency by calculating the average number of
posts per day since account creation
    """
    if posts and account_age_days > 0:
        posts_per_day = len(posts) / max(account_age_days, 1)

        # Normal users rarely post more than 10 times per day consistently
        if posts_per_day > 10:
            suspicious_activity_score += 0.5

        # Accounts posting 20+ times per day are almost certainly automated
bots
        if posts_per_day > 20:
            suspicious_activity_score += 0.5

    user_risk_score += suspicious_activity_score

    # Step 6
    final_score = min(5.0, user_risk_score)

    return final_score
```

Top 5 highest risk users:

ID	Username	Profile Bio	Created At	Risk Level	Actions
11	ChilliPepper	Parent. Gamer. Nature lover. Night owl. Proudly from Mexico City. Some…	Jul 14, 2021 05:45	HIGH (5.0)	Delete
13	StarGazer	Dreamer of worlds, in pages I find, \nMelodies soothe my restless mind. …	Feb 25, 2021 16:00	HIGH (5.0)	Delete
18	SilverSurfer	Dreamer. Lifelong learner. Parenting enthusiast. Coffee lover. Pet whisp…	May 30, 2021 08:45	HIGH (5.0)	Delete
66	BlackPearl	Fashionista. Pet Lover. Sports Enthusiast. Dreamer. Overthinker. Alway…	Sep 27, 2021 13:30	HIGH (5.0)	Delete
68	SilverFox	\"'Level up in life, one rep at a time.' Fitness junkie, meme connoisseur, …	May 28, 2021 09:30	HIGH (5.0)	Delete

Exercise 3.3 Recommendation Algorithm: Implement the `recommend` function. Identify a suitable, simple recommendation algorithm that will recommend 5 relevant posts on the "Recommended" tab based on the posts the user reacted to positively and the users they followed. (5 points)

```
def recommend(user_id, filter_following):
    """
    Args:
        user_id: The ID of the current user.
        filter_following: Boolean, True if we only want to see recommendations
from followed users.

    Returns:
        A list of 5 recommended posts, in reverse-chronological order.

    To test whether your recommendation algorithm works, let's pretend we like
the DIY topic. Here are some users that often post DIY comment and a few
example posts. Make sure your account did not engage with anything else. You
should test your algorithm with these and see if your recommendation algorithm
picks up on your interest in DIY and starts showing related content.

    Users: @starboy99, @DancingDolphin, @blogger_bob
    Posts: 1810, 1875, 1880, 2113

    Materials:
    - https://www.nvidia.com/en-us/glossary/recommendation-system/
    - http://www.configworks.com/mz/handout_recsys_sac2010.pdf
    -
https://www.researchgate.net/publication/227268858_Recommender_Systems_Handbook

    After reading through the materials, I decided to implement a hybrid
recommendation system that combines content-based filtering with collaborative
filtering, explicitly weighting different types of user feedback, and improving
cold start handling.
```

```
    Besides, I also implemented several NLP techniques:
    - Stop word filtering
    - TF weighting
    - User similarity via collaborative filtering

    After tried to follow Users with DIY interests and react to their posts,
the recommendation algorithm started to show more DIY-related posts in the
recommend tab.
    """

    # Cold Start Strategy
    """
    If the user is not logged in, I simply return the 5 most recent posts by
selecting from the posts table ordered by created_at DESC
    """
    if not user_id:
        recent_posts = query_db('''
            SELECT p.id, p.content, p.created_at, u.username, u.id as user_id
            FROM posts p
            JOIN users u ON p.user_id = u.id
            ORDER BY p.created_at DESC
            LIMIT 5
        ''')
        return recent_posts if recent_posts else []

    # Check if user has interactions
    """
    This query checks if the user has any reactions recorded in the reactions
table
    The WHERE r.user_id = ? clause get the user_id that are passed into and
filters reactions to only those made by the current user
    """
    user_reactions = query_db('''
        SELECT p.content, r.reaction_type
        FROM reactions r
        JOIN posts p ON r.post_id = p.id
        WHERE r.user_id = ?
    ''', (user_id,))

    if not user_reactions:
        if filter_following:
            """
            This query fetch the most recent posts from users that the current
user follows by joining the posts, users, and follows tables. The WHERE
```

```
f.follower_id = ? clause filters the posts to only those made by users that the
current user follows
            """
            qr = query_db('''
                SELECT DISTINCT p.id, p.content, p.created_at, u.username, u.id
as user_id
                FROM posts p
                JOIN users u ON p.user_id = u.id
                JOIN follows f ON p.user_id = f.followed_id
                WHERE f.follower_id = ? AND p.user_id != ?
                ORDER BY p.created_at DESC
                LIMIT 5
            ''', (user_id, user_id))
        else:
            """
            This query fetches the 5 most recent posts from all users except
the current user (WHERE p.user_id != ?) by joining the posts and users tables
            """
            qr = query_db('''
                SELECT p.id, p.content, p.created_at, u.username, u.id as
user_id
                FROM posts p
                JOIN users u ON p.user_id = u.id
                WHERE p.user_id != ?
                ORDER BY p.created_at DESC
                LIMIT 5
            ''', (user_id,))
        return qr if qr else []

    """
    I decided to assign different weights to different reaction types to
reflect their significance in indicating user interest
    """
    REACTION_WEIGHTS = {
        'love': 2.0, 'like': 1.5, 'wow': 1.2,
        'laugh': 1.0, 'sad': 0.3, 'angry': 0.1
    }

    """
    To find interest keywords, I analyze the content of posts the user has
reacted to, applying weights based on reaction types. I also implement stop
word filtering to focus on meaningful keywords and give more weight to hashtags
    """
    interest_keywords = {}
```

```
    for reaction in user_reactions:
        weight = REACTION_WEIGHTS.get(reaction['reaction_type'], 0.5)
        words = reaction['content'].lower().split()

        for word in words:
            clean_word = ''.join(c for c in word if c.isalnum() or c == '#')
            # Stop word filtering
            if len(clean_word) >= 3 and clean_word.lower() not in STOP_WORDS:
                if clean_word.startswith('#'):
                    weight *= 2  # Hashtags are strong signals
                interest_keywords[clean_word] =
interest_keywords.get(clean_word, 0) + weight

    """
    This query identifies users with same interest by finding common reactions
on the same posts. It count the number of common likes between the current user
and other users, filtering for those with at least 2 common likes by joining
the reactions table on itself and than grouping by the other user's ID. 5
similar users will be selected based on the highest count of common likes
    """
    similar_users = query_db('''
        SELECT r2.user_id, COUNT(*) as common_likes
        FROM reactions r1
        JOIN reactions r2 ON r1.post_id = r2.post_id
        WHERE r1.user_id = ? AND r2.user_id != ?
        GROUP BY r2.user_id
        HAVING common_likes >= 2
        ORDER BY common_likes DESC
        LIMIT 5
    ''', (user_id, user_id))

    similar_user_ids = [u['user_id'] for u in similar_users] if similar_users
else []

    # Exclude those from recommendations)
    reacted_post_ids = query_db('''
        SELECT post_id FROM reactions WHERE user_id = ?
    ''', (user_id,))

    # This is the react ids of the posts the user has already reacted to
    reacted_ids = [str(row['post_id']) for row in reacted_post_ids] if
reacted_post_ids else []

    """
```

```
    I fetch candidate posts based on whether to filter by followed users or
not, and exclude posts the user has already reacted to. The queries join the
posts and users tables, and order the results by creation date to prioritize
recent content

    The flow for this section is as follows:
    - If filter_following is True:
        - If reacted_ids is not empty, fetch posts from followed users
excluding reacted posts
        - Else, fetch posts from followed users
    - Else:
        - If reacted_ids is not empty, fetch posts from all users excluding
reacted posts
        - Else, fetch posts from all users
    """
    if filter_following:
        if reacted_ids:
            candidates = query_db('''
                SELECT DISTINCT p.id, p.content, p.created_at, u.username, u.id
as user_id
                FROM posts p
                JOIN users u ON p.user_id = u.id
                JOIN follows f ON p.user_id = f.followed_id
                WHERE f.follower_id = ? AND p.user_id != ?
                  AND p.id NOT IN ({})
                ORDER BY p.created_at DESC
                LIMIT 100
            '''.format(','.join('?' * len(reacted_ids))), (user_id, user_id) +
tuple(reacted_ids))
        else:
            candidates = query_db('''
                SELECT DISTINCT p.id, p.content, p.created_at, u.username, u.id
as user_id
                FROM posts p
                JOIN users u ON p.user_id = u.id
                JOIN follows f ON p.user_id = f.followed_id
                WHERE f.follower_id = ? AND p.user_id != ?
                ORDER BY p.created_at DESC
                LIMIT 100
            ''', (user_id, user_id))
    else:
        if reacted_ids:
            candidates = query_db('''
```

```
                SELECT p.id, p.content, p.created_at, u.username, u.id as
user_id
                FROM posts p
                JOIN users u ON p.user_id = u.id
                WHERE p.user_id != ?
                  AND p.id NOT IN ({})
                ORDER BY p.created_at DESC
                LIMIT 200
            '''.format(','.join('?' * len(reacted_ids))), (user_id,) +
tuple(reacted_ids))
        else:
            candidates = query_db('''
                SELECT p.id, p.content, p.created_at, u.username, u.id as
user_id
                FROM posts p
                JOIN users u ON p.user_id = u.id
                WHERE p.user_id != ?
                ORDER BY p.created_at DESC
                LIMIT 200
            ''', (user_id, user_id))

    if not candidates:
        return []

    scored_posts = []

    for post in candidates:
        score = 0

        """
        Content-Based Filtering
        I analyze the content of each candidate post for keywords that match
the user's interests, increamenting the score based on the presence and weight
of these keywords
        """
        post_words = post['content'].lower().split()
        for word in post_words:
            clean_word = ''.join(c for c in word if c.isalnum() or c == '#')
            if clean_word in interest_keywords:
                score += interest_keywords[clean_word]

        """
        Collaborative Filtering
```

```
        I check if any similar users have liked the candidate post. If so, I
increase the score
        """
        if similar_user_ids:
            for similar_user in similar_user_ids:
                liked_by_similar = query_db('''
                    SELECT 1 FROM reactions
                    WHERE post_id = ? AND user_id = ?
                    LIMIT 1
                ''', (post['id'], similar_user), one=True)
                if liked_by_similar:
                    score += 2

        """
        I also increase the score for more recent posts to prioritize fresh
content
        """
        post_date = post['created_at'] if isinstance(post['created_at'],
datetime) else datetime.strptime(post['created_at'], '%Y-%m-%d %H:%M:%S')
        days_old = (datetime.utcnow() - post_date).days
        if days_old < 7:
            score += 1
        elif days_old < 30:
            score += 0.5

        scored_posts.append((post, score))

    scored_posts.sort(key=lambda x: x[1], reverse=True)
    top_posts = [post for post, score in scored_posts[:5]]

    top_posts.sort(key=lambda x: x['created_at'], reverse=True)

    return top_posts
```

Task 4 (due 27.10.2025 23:59) 20 points

Exercise 4.1 Topics: Identify the 10 most popular topics discussed on our platform. Use Latent Dirichlet Allocation (LDA) with the `gensim` library. Answer and explain your queries/calculations below. You may use SQL and/or Python to perform this task. 5 points)

```
# Explainations for the work are being added as comments
import sqlite3
import pandas as pd
import re
import nltk
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer
from nltk.tokenize import word_tokenize
from gensim import corpora
from gensim.models import LdaModel, CoherenceModel
from collections import Counter  # More efficient for counting
import warnings
warnings.filterwarnings('ignore')

"""
After researching different approaches for determining the optimal number of
topics, I chose to use Coherence Score Optimization with grid search. There is
actually different methods, Hierarchical Dirich Process is one of them, but it
could lead to less interpretable and unstable results.

The coherence score (c_v) is the best metric for evaluating topic models
because it measures how semantically related the words in each topic are.
Higher coherence indicates more interpretable and distinct topics. This is
widely used in research papers. For our dataset of 1303 posts, testing K from 5
to 20 is reasonable and fast because it covers both broad themes and specific
topics. It also provides a clear, explainable methodology.

Steps:
- Extract all posts from the database using SQL
- Preprocess the text (cleaning, tokenization, lemmatization, stopword removal)
- Create a dictionary and corpus for LDA
- Test different K values (5-20) and calculate coherence score for each
- Select the K with the highest coherence score as optimal
- Train the final LDA model with optimal K
- Identify and rank the top 10 topics by number of posts
"""
```

```python
# Download required NLTK data
try:
    nltk.data.find('corpora/stopwords')
    nltk.data.find('corpora/wordnet')
    nltk.data.find('tokenizers/punkt')
except LookupError:
    nltk.download('stopwords', quiet=True)
    nltk.download('wordnet', quiet=True)
    nltk.download('punkt', quiet=True)
    nltk.download('omw-1.4', quiet=True)

# Current db file location
dbfile = 'database.sqlite'
# Establish a connection to the db
conn = sqlite3.connect(dbfile)

def preprocess_text(text):
    """
    Preprocess text: lowercase, remove URLs/special chars, tokenize, remove
stopwords, lemmatize
    """
    text = text.lower()
    text = re.sub(r'http\S+|www\S+|https\S+', '', text, flags=re.MULTILINE)  #
Remove URLs
    text = re.sub(r'@\w+', '', text)  # Remove mentions
    text = re.sub(r'#(\w+)', r'\1', text)  # Keep hashtag words
    text = re.sub(r'[^a-z\s]', '', text)  # Keep only letters

    tokens = word_tokenize(text)

    # Remove stopwords and short tokens
    stop_words = set(stopwords.words('english'))

    """
    I add some custom stopwords that are too generic and do not contribute to
topic meaning
    """
    custom_stopwords = {
        'today', 'day', 'time', 'got', 'went', 'made', 'make', 'tried', 'try',
'started', 'spent', 'new', 'old', 'still', 'finally', 'just', 'perfect',
'amazing', 'bad', 'like', 'love', 'really', 'very', 'always', 'never', 'thing',
'things', 'something', 'anything', 'people', 'actually', 'basically',
'literally', 'damn', 'now', 'then', 'ago'
    }
```

```
    stop_words.update(custom_stopwords)

    tokens = [token for token in tokens if token not in stop_words and
len(token) > 2]

    # Lemmatize to reduce words to base form
    lemmatizer = WordNetLemmatizer()
    tokens = [lemmatizer.lemmatize(token) for token in tokens]

    return tokens

def load_and_preprocess_posts():
    """Load posts from database and preprocess them"""
    posts_df = pd.read_sql_query("SELECT id, content FROM posts WHERE content
IS NOT NULL AND content != '';", conn)
    print(f"Loaded {len(posts_df)} posts from database")

    # Invoke preprocess function using apply
    posts_df['processed_tokens'] = posts_df['content'].apply(preprocess_text)
    return posts_df

def generate_topic_label(top_words):
    """
    Generate an interpretable label for a topic based on its top words.

    NOTE: Since LDA only produces word distributions without semantic labels,
topic labels were assigned manually based on the top 10 words for each topic,
following standard practice in topic modeling research. I chose this approach
because automated labeling methods often lack the nuance and context that human
interpretation provides. At the end, those methods are based on what humans
understand from the words.

    The function uses keyword matching to suggest labels, but these should be
reviewed and adjusted based on domain knowledge and context.
    """
    keywords_map = {
        'Food & Cooking': ['recipe', 'cooking', 'cook', 'chef', 'baking',
'meal', 'dish', 'dinner', 'lunch', 'ingredient', 'kitchen', 'delicious',
'tasty', 'flavor', 'eat', 'food'],
        'Fitness & Health': ['workout', 'fitness', 'gym', 'exercise',
'training', 'cardio', 'muscle', 'strength', 'running', 'weight', 'yoga',
'reps', 'protein', 'health', 'fit'],
```

```
        'Mental Health': ['mental', 'therapy', 'anxiety', 'depression',
'mindfulness', 'wellness', 'selfcare', 'healing', 'counseling',
'mentalhealth'],
        'Books & Reading': ['book', 'read', 'reading', 'novel', 'author',
'library', 'literature', 'chapter', 'page', 'fiction', 'story', 'bookclub',
'bookworm', 'writing', 'meme'],
        'Nature & Outdoors': ['hiking', 'trail', 'mountain', 'forest',
'wilderness', 'camping', 'wildlife', 'backpacking', 'naturelover', 'scenic',
'peak', 'outdoor', 'adventure', 'nature'],
        'Technology & Gaming': ['tech', 'game', 'gaming', 'gamer', 'console',
'gameplay', 'streamer', 'coding', 'programming', 'software', 'computer',
'digital', 'esports', 'technology'],
        'Politics & News': ['politics', 'political', 'government', 'election',
'vote', 'policy', 'legislation', 'campaign', 'senate', 'congress', 'democrat',
'republican', 'debate'],
        'DIY & Crafts': ['diy', 'craft', 'crafting', 'handmade', 'woodworking',
'sewing', 'knitting', 'pottery', 'maker', 'upcycling', 'tutorial', 'project',
'build', 'building'],
        'Travel & Photography': ['travel', 'trip', 'vacation', 'destination',
'wanderlust', 'photography', 'photographer', 'camera', 'lens', 'passport',
'tourist', 'journey', 'explore'],
        'Community & Social': ['community', 'volunteer', 'charity',
'nonprofit', 'activism', 'neighbor', 'civic', 'volunteering', 'local',
'outreach', 'fundraiser', 'together', 'support'],
        'Climate & Environment': ['climate', 'climatechange', 'environmental',
'sustainability', 'renewable', 'carbon', 'emissions', 'conservation',
'pollution', 'green', 'ecofriendly', 'environment', 'change'],
        'Fashion & Style': ['fashion', 'style', 'outfit', 'fashionista',
'trendy', 'wardrobe', 'designer', 'clothing', 'wear', 'dress', 'aesthetic',
'ootd', 'look'],
        'Art & Creativity': ['art', 'artist', 'painting', 'drawing',
'sculpture', 'gallery', 'canvas', 'illustration', 'sketch', 'artistic',
'creative', 'music', 'design'],
        'Entertainment & Media': ['movie', 'film', 'series', 'streaming',
'netflix', 'show', 'episode', 'watch', 'cinema', 'actor', 'hollywood', 'video',
'funny', 'meme', 'best'],
        'Personal Growth': ['growth', 'mindset', 'motivation', 'inspiration',
'goals', 'achievement', 'learning', 'wisdom', 'progress', 'perspective',
'transformation', 'development', 'real', 'energy'],
        'Feelings & Emotions': ['feeling', 'feel', 'felt', 'like', 'love',
'hate', 'happy', 'sad', 'mood', 'emotion'],
        'Daily Moments': ['today', 'simple', 'moment'],
        'Routine & Habits': ['day', 'classic', 'routine', 'daily', 'morning',
'coffee', 'breakfast'],
```

```
        'Current State': ['still', 'now', 'currently'],
        'Quality & Perfection': ['perfect', 'great', 'awesome', 'amazing'],
        'Time Spent': ['time', 'spent', 'hour', 'week', 'weekend', 'year',
'ago'],
        'Trying Things': ['tried', 'try', 'attempt', 'testing'],
        'Getting Things': ['got', 'bought', 'received', 'latest'],
        'Completing Things': ['finished', 'completed', 'done', 'finally'],
        'New Experiences': ['new', 'first', 'discover', 'fresh', 'another',
'recent'],
        'Social Observations': ['people', 'everyone', 'someone', 'anyone',
'damn', 'guy', 'person', 'friend'],
        'Life Philosophy': ['life', 'never', 'always', 'think', 'thought'],
        'Understanding': ['knew', 'know', 'understand', 'realize'],
    }

    # Count matches for each category with better scoring
    category_scores = []

    """
    Here I'm using a weighted scoring system to prioritize top words more
heavily. The top 3 words get a weight of 3, the next 2 words get a weight of 2,
and the remaining words get a weight of 1. This way, if a category matches with
the most significant words of the topic, it will score higher and be more
likely to be selected as the label
    """
    for category, keywords in keywords_map.items():
        score = 0
        for i, word in enumerate(top_words[:10]):
            if word in keywords:
                weight = 3 if i < 3 else (2 if i < 5 else 1)  # Higher weight
for top words
                score += weight
        if score > 0:
            category_scores.append((category, score))

    # Get best match
    if category_scores:
        # Sort by score and avoid duplicates by checking if label was already
used
        category_scores.sort(key=lambda x: x[1], reverse=True)
        best_match = category_scores[0][0]
    else:
        # Fallback: create descriptive label from top 3 words
        best_match = f"Topic: {', '.join(top_words[:3])}"
```

```
    return best_match

def create_dictionary_and_corpus(documents):
    """Create dictionary and corpus (bag of words) for LDA"""
    dictionary = corpora.Dictionary(documents)

    # Filter extremes: remove words that appear in <5 documents or >50% of
documents
    dictionary.filter_extremes(no_below=5, no_above=0.5)
    print(f"Dictionary created with {len(dictionary)} terms")

    corpus = [dictionary.doc2bow(doc) for doc in documents]
    return dictionary, corpus

def find_optimal_k(corpus, dictionary, documents, k_range=range(2, 21)):
    """
    Find the optimal number of topics K by testing different values and
comparing coherence scores.

    Coherence score measures how interpretable the topics are. Higher coherence
means the words in each topic are more semantically related, making topics more
meaningful and distinct.

    I use c_v coherence as it correlates well with human topic interpretability
judgments.

    c_v is based on a sliding window, a one-set segmentation of the top words,
and an indirect confirmation measure that uses normalized pointwise mutual
information and the cosine similarity.
    """
    print("\nFinding optimal K by testing different numbers of topics...")
    print(f"Testing K values from {min(k_range)} to {max(k_range)}")
    print(f"{'K':<5} {'Coherence Score':<20}")
    print("-" * 25)

    coherence_scores = []

    for k in k_range:
        # Train LDA model with k topics
        lda_model = LdaModel(
            corpus=corpus,
            id2word=dictionary,
            num_topics=k,
```

```
            random_state=42,
            passes=10,  # Reduced for faster testing
            iterations=200,
            alpha='auto',
            eta='auto',
            per_word_topics=True
        )

        # Calculate coherence score
        coherence_model = CoherenceModel(
            model=lda_model, texts=documents, dictionary=dictionary,
coherence='c_v'
        )
        coherence = coherence_model.get_coherence()
        coherence_scores.append((k, coherence))
        print(f"{k:<5} {coherence:<20.4f}")

    # Find K with highest coherence
    optimal_k, best_coherence = max(coherence_scores, key=lambda x: x[1])
    print(f"\nOptimal K = {optimal_k} (Coherence: {best_coherence:.4f})")

    return optimal_k, coherence_scores

def train_lda_model(corpus, dictionary, num_topics):
    """Train final LDA model with optimal number of topics"""
    print(f"\nTraining final LDA model with K={num_topics} topics...")

    lda_model = LdaModel(
        corpus=corpus,
        id2word=dictionary,
        num_topics=num_topics,
        random_state=42,
        passes=15,  # More passes for final model
        iterations=400,
        alpha='auto',  # Auto-learn document-topic density
        eta='auto',  # Auto-learn topic-word density
        per_word_topics=True
    )

    print("Model training completed")
    return lda_model

def evaluate_model(lda_model, corpus, dictionary, documents):
    """
```

```
    Evaluate final model quality using coherence score and perplexity.

    Coherence measures topic interpretability (higher is better).
    Perplexity measures how well the model predicts the data (lower is better).
    """
    # Calculate coherence score
    coherence_model = CoherenceModel(
        model=lda_model, texts=documents, dictionary=dictionary,
coherence='c_v'
    )
    coherence_score = coherence_model.get_coherence()

    # Calculate perplexity
    perplexity = lda_model.log_perplexity(corpus)

    print(f"Final Model Coherence Score: {coherence_score:.4f} (higher is
better)")
    print(f"Final Model Perplexity: {perplexity:.4f} (lower is better)")

    return coherence_score, perplexity

def extract_and_display_topics(lda_model, num_topics, num_words=10):
    """Extract and display all topics with their top words"""
    topics_data = []

    for idx, topic in lda_model.print_topics(num_topics=num_topics,
num_words=num_words):
        words_weights = []
        for item in topic.split(' + '):
            weight, word = item.split('*')
            word = word.strip('"')
            weight = float(weight)
            words_weights.append((word, weight))

        top_words = [word for word, weight in words_weights]

        # Generate interpretable topic label based on top words
        topic_label = generate_topic_label(top_words)

        topics_data.append({
            'topic_id': idx,
            'label': topic_label,
            'words': words_weights,
            'top_words': top_words
```

```
        })

    # Fix duplicate labels by adding distinguishing words
    # This is because I detected some topics had same labels in the output
    label_counts = Counter(t['label'] for t in topics_data)
    for topic in topics_data:
        if label_counts[topic['label']] > 1:
            # Add top 2 distinctive words to make label unique
            topic['label'] = f"{topic['label']}: {',
'.join(topic['top_words'][:2])}"

    return topics_data

def analyze_topic_distribution(lda_model, corpus, num_topics, topics_data):
    """
    Analyze topic distribution and identify the TOP 10 topics with the most
posts.

    For this, I used Counter instead of dict + manual counting for better
performance. Counter.most_common() uses a heap internally, which is more
efficient than sorting all items when you only need the top K.
    """
    print(f"\n{'='*80}")
    print("TOPIC DISTRIBUTION (All Topics)")
    print(f"{'='*80}\n")

    topic_counter = Counter()
    for doc_topics in lda_model.get_document_topics(corpus):
        if doc_topics:
            dominant_topic = max(doc_topics, key=lambda x: x[1])[0]
            topic_counter[dominant_topic] += 1

    total = len(corpus)
    for topic_id in range(num_topics):
        count = topic_counter.get(topic_id, 0)  # Default to 0 if topic has no
posts
        pct = (count / total) * 100
        label = topics_data[topic_id]['label']
        print(f"Topic {topic_id + 1} ({label}): {count} posts ({pct:.1f}%)")

    # Identify TOP 10 topics by post count
    print(f"\n{'='*80}")
    print("TOP 10 TOPICS WITH MOST POSTS (Answer to Exercise 4.1)")
    print(f"{'='*80}\n")
```

```
    # most_common(10) uses heap internally - O(n log k)
    top_10_topics = topic_counter.most_common(10)

    print(f"{'Rank':<6} {'Topic':<10} {'Topic Name':<30} {'Posts':<10}
{'%':<8}")
    print("-" * 70)
    for rank, (topic_id, count) in enumerate(top_10_topics, 1):
        pct = (count / total) * 100
        label = topics_data[topic_id]['label']
        print(f"{rank:<6} Topic {topic_id + 1:<3} {label:<30} {count:<10}
{pct:>6.1f}%")

    return topic_counter, top_10_topics

def main():
    # Load and preprocess posts
    posts_df = load_and_preprocess_posts()
    documents = posts_df['processed_tokens'].tolist()

    # Create dictionary and corpus
    dictionary, corpus = create_dictionary_and_corpus(documents)

    # Find optimal K by comparing coherence scores
    # I chose range 5-20 based on dataset size and diversity
    # It can cover both broad themes and specific topics
    optimal_k, coherence_scores = find_optimal_k(corpus, dictionary, documents,
k_range=range(5, 21))

    # Train final LDA model with optimal K
    lda_model = train_lda_model(corpus, dictionary, num_topics=optimal_k)

    # Save model and dictionary for use in task 4.2
    lda_model.save('lda_model_k20.model')
    dictionary.save('lda_dictionary.dict')

    # Evaluate final model
    coherence_score, perplexity = evaluate_model(lda_model, corpus, dictionary,
documents)

    # Extract and display all topics
    topics_data = extract_and_display_topics(lda_model, num_topics=optimal_k,
num_words=10)
```

```
    # Analyze distribution and identify TOP 10 topics with most posts
    analyze_topic_distribution(lda_model, corpus, num_topics=optimal_k,
topics_data=topics_data)

if __name__ == "__main__":
    try:
        main()
    except Exception as e:
        print(f"Error: {e}")
    finally:
        conn.close()

"""
Loaded 1303 posts from database
Dictionary created with 685 terms

Finding optimal K by testing different numbers of topics...
Testing K values from 5 to 20
K     Coherence Score
-------------------------
5     0.3700
6     0.3759
7     0.4289
8     0.4130
9     0.3971
10    0.4312
11    0.3854
12    0.4132
13    0.4089
14    0.4230
15    0.4187
16    0.3891
17    0.3877
18    0.4053
19    0.4114
20    0.3980

# This optimal K could be affected by the change of custom stopwords or
preprocessing steps
Optimal K = 10 (Coherence: 0.4312)

Training final LDA model with K=10 topics...
Model training completed
Final Model Coherence Score: 0.4152 (higher is better)
```

```
Final Model Perplexity: -6.4902 (lower is better)

================================================================================
=
TOPIC DISTRIBUTION (All Topics)
================================================================================
=

Topic 1 (Understanding): 131 posts (10.1%)
Topic 2 (Feelings & Emotions): 146 posts (11.2%)
Topic 3 (Life Philosophy): 115 posts (8.8%)
Topic 4 (Personal Growth): 109 posts (8.4%)
Topic 5 (Fitness & Health): 154 posts (11.8%)
Topic 6 (Books & Reading): 120 posts (9.2%)
Topic 7 (Entertainment & Media): 102 posts (7.8%)
Topic 8 (DIY & Crafts): 112 posts (8.6%)
Topic 9 (Politics & News): 145 posts (11.1%)
Topic 10 (Nature & Outdoors): 169 posts (13.0%)

================================================================================
=
TOP 10 TOPICS WITH MOST POSTS (Answer to Exercise 4.1)
================================================================================
=

Rank   Topic      Topic Name                    Posts      %
----------------------------------------------------------------------
1      Topic 10  Nature & Outdoors            169         13.0%
2      Topic 5   Fitness & Health             154         11.8%
3      Topic 2   Feelings & Emotions          146         11.2%
4      Topic 9   Politics & News              145         11.1%
5      Topic 1   Understanding                131         10.1%
6      Topic 6   Books & Reading              120          9.2%
7      Topic 3   Life Philosophy              115          8.8%
8      Topic 8   DIY & Crafts                 112          8.6%
9      Topic 4   Personal Growth              109          8.4%
10     Topic 7   Entertainment & Media        102          7.8%
"""
```

Exercise 4.2 Sentiment: Perform sentiment analysis on posts and comments. What is the overall tone of the platform? How does sentiment vary across user posts discussing different topics identified in Exercise 4.1? Please use VADER (`nltk.sentiment`) for this analysis. Answer and explain your queries/calculations below. You may use SQL and/or Python to perform this task. (5 points)

```python
# Explainations for the work are being added as comments
import sqlite3
import pandas as pd
import numpy as np
import nltk
from nltk.sentiment import SentimentIntensityAnalyzer
import warnings
import matplotlib.pyplot as plt
import seaborn as sns
# Import preprocess_text and generate_topic_label from task1
from task1 import preprocess_text, generate_topic_label
from gensim.models import LdaModel
from gensim import corpora

warnings.filterwarnings('ignore')

"""
For this analysis, I will:
- Analyze sentiment of all posts and comments using VADER
- Calculate overall platform sentiment (compound scores)
- Link posts to topics from Exercise 4.1 (import it) using the same LDA model
- Compare sentiment across different topics
- Provide insights on which topics have more positive/negative sentiment
"""

# Download VADER lexicon
try:
    nltk.data.find('sentiment/vader_lexicon.zip')
except LookupError:
    nltk.download('vader_lexicon', quiet=True)

# Database connection
dbfile = 'database.sqlite'
# Establish a connection to the db
conn = sqlite3.connect(dbfile)

sia = SentimentIntensityAnalyzer()

def analyze_sentiment_batch(df, content_column='content'):
    """
    Analyze sentiment using VADER and return all scores.

    VADER returns sentiment_dict, which includes:
```

```
    - compound: Overall sentiment (-1 most negative to +1 most positive)
    - pos: Positive sentiment proportion
    - neu: Neutral sentiment proportion
    - neg: Negative sentiment proportion

    The compound score interpretation:
    - >= 0.05: Positive
    - <= -0.05: Negative
    - Between -0.05 and 0.05: Neutral

    I choose these points because they are standard thresholds
    """

    # Generic function for both posts and comments
    sentiment_results = df[content_column].apply(lambda x:
sia.polarity_scores(x))

    df['compound'] = sentiment_results.apply(lambda x: x['compound'])
    df['positive'] = sentiment_results.apply(lambda x: x['pos'])
    df['neutral'] = sentiment_results.apply(lambda x: x['neu'])
    df['negative'] = sentiment_results.apply(lambda x: x['neg'])
    df['sentiment_category'] = df['compound'].apply(categorize_sentiment)

    return df

def categorize_sentiment(compound_score):
    """Categorize sentiment based on compound score"""
    if compound_score >= 0.05:
        return 'Positive'
    elif compound_score <= -0.05:
        return 'Negative'
    else:
        return 'Neutral'

def analyze_posts_sentiment():
    # Load all posts that have content
    posts_df = pd.read_sql_query("""
        SELECT id, user_id, content, created_at
        FROM posts
        WHERE content IS NOT NULL AND content != ''
    """, conn)

    print(f"Analyzing {len(posts_df)} posts...")
```

```python
    return analyze_sentiment_batch(posts_df)

def analyze_comments_sentiment():
    # Load all comments that have content
    comments_df = pd.read_sql_query("""
        SELECT id, post_id, user_id, content, created_at
        FROM comments
        WHERE content IS NOT NULL AND content != ''
    """, conn)

    print(f"Analyzing {len(comments_df)} comments...")

    return analyze_sentiment_batch(comments_df)

def display_overall_sentiment(posts_df, comments_df):
    # Posts sentiment summary
    print("\n--- POSTS SENTIMENT ---")
    print(f"Average compound score: {posts_df['compound'].mean():.4f}")
    print(f"Median compound score: {posts_df['compound'].median():.4f}")
    print(f"Std deviation: {posts_df['compound'].std():.4f}")

    print(f"\nSentiment distribution:")
    sentiment_counts = posts_df['sentiment_category'].value_counts()
    for category in ['Positive', 'Neutral', 'Negative']:
        count = sentiment_counts.get(category, 0)
        pct = (count / len(posts_df)) * 100
        print(f"  {category}: {count} ({pct:.1f}%)")

    # Comments sentiment summary
    print("\n--- COMMENTS SENTIMENT ---")
    print(f"Average compound score: {comments_df['compound'].mean():.4f}")
    print(f"Median compound score: {comments_df['compound'].median():.4f}")
    print(f"Std deviation: {comments_df['compound'].std():.4f}")

    print(f"\nSentiment distribution:")
    sentiment_counts = comments_df['sentiment_category'].value_counts()
    for category in ['Positive', 'Neutral', 'Negative']:
        count = sentiment_counts.get(category, 0)
        pct = (count / len(comments_df)) * 100
        print(f"  {category}: {count} ({pct:.1f}%)")

    # Overall platform tone
    print("\n--- OVERALL PLATFORM TONE ---")
    """
```

```python
    I calculate the overall platform tone by averaging the compound scores
    from both posts and comments
    """
    all_compound = pd.concat([posts_df['compound'], comments_df['compound']])
    avg_compound = all_compound.mean()

    if avg_compound >= 0.05:
        tone = "POSITIVE"
    elif avg_compound <= -0.05:
        tone = "NEGATIVE"
    else:
        tone = "NEUTRAL"

    print(f"Average compound score (all content): {avg_compound:.4f}")
    print(f"Overall platform tone: {tone}")

    if avg_compound > 0:
        print(f"\nThe platform has a slightly positive tone")
    else:
        print(f"\nThe platform has a neutral to slightly negative tone.")

def assign_topics_to_posts(posts_df):
    """
    Assign topics to posts using the LDA model from Exercise 4.1.
    This reuses the preprocessing and LDA model to ensure consistency.
    """
    print("--- Assigning topics to posts using LDA model ---")

    print("\nPreprocessing posts...")
    posts_df['processed_tokens'] = posts_df['content'].apply(preprocess_text)
    documents = posts_df['processed_tokens'].tolist()

    print("Loading LDA model and dictionary from disk...")
    lda_model = LdaModel.load('lda_model_k20.model')
    dictionary = corpora.Dictionary.load('lda_dictionary.dict')

    print("Creating corpus using loaded dictionary...")
    corpus = [dictionary.doc2bow(tokens) for tokens in documents]

    # Assign dominant topic to each post
    print("Assigning topics to posts...")
    topics = []
    for doc_topics in lda_model.get_document_topics(corpus):
        if doc_topics:
```

```
            dominant_topic = max(doc_topics, key=lambda x: x[1])[0]
            topics.append(dominant_topic)
        else:
            topics.append(-1)  # No topic assigned

    posts_df['topic_id'] = topics

    topic_labels = {}
    topic_keywords = {}  # Store keywords for display
    for idx in range(10):
        topic_words = lda_model.show_topic(idx, topn=10)
        words = [word for word, _ in topic_words]
        topic_keywords[idx] = ', '.join(words[:3])  # Keep top 3 for reference

        # Use the same labeling logic from task1
        label = generate_topic_label(words)
        topic_labels[idx] = label

    # Assign topic labels and keywords
    posts_df['topic_label'] = posts_df['topic_id'].apply(lambda x:
topic_labels.get(x, 'Unknown'))
    posts_df['topic_keywords'] = posts_df['topic_id'].apply(lambda x:
topic_keywords.get(x, ''))

    print(f"Successfully assigned topics to {len(posts_df[posts_df['topic_id']
!= -1])} posts")

    return posts_df, topic_labels, topic_keywords

def analyze_sentiment_by_topic(posts_df, topic_labels, topic_keywords):
    # Filter out posts without topics
    posts_with_topics = posts_df[posts_df['topic_id'] != -1].copy()

    # Group by topic and calculate sentiment statistics
    topic_sentiment = posts_with_topics.groupby('topic_id').agg({
        'compound': ['mean', 'median', 'std', 'count'],
        'positive': 'mean',
        'negative': 'mean',
        'neutral': 'mean'
    }).round(4)

    # Flatten column names
    topic_sentiment.columns = ['_'.join(col).strip() for col in
topic_sentiment.columns.values]
```

```
    topic_sentiment = topic_sentiment.reset_index()

    # Add topic labels and keywords
    topic_sentiment['topic_name'] = topic_sentiment['topic_id'].apply(lambda x:
topic_labels.get(x, 'Unknown'))
    topic_sentiment['keywords'] = topic_sentiment['topic_id'].apply(lambda x:
topic_keywords.get(x, ''))

    # Sort by average compound score
    topic_sentiment = topic_sentiment.sort_values('compound_mean',
ascending=False)

    print(f"\nSentiment Summary by Topic (sorted by average compound score):")
    print(f"{'Rank':<6} {'Topic Name':<35} {'Avg Score':<12} {'Posts':<8}
{'Pos%':<8} {'Neg%':<8}")
    print("-"*80)

    for rank, (_, row) in enumerate(topic_sentiment.iterrows(), 1):
        topic_no = row['topic_id']
        topic_name = row['topic_name']
        avg_score = row['compound_mean']
        post_count = int(row['compound_count'])
        pos_pct = row['positive_mean'] * 100
        neg_pct = row['negative_mean'] * 100

        topic_display = f"{topic_name}"
        print(f"{rank:<6} {topic_display:<35} {avg_score:<12.4f}
{post_count:<8} {pos_pct:<8.1f} {neg_pct:<8.1f}")

    # Detailed analysis for top 3 most positive and negative topics
    print("\nTOP 3 MOST POSITIVE TOPICS")

    top_positive = topic_sentiment.head(3)
    for idx, (_, row) in enumerate(top_positive.iterrows(), 1):
        topic_name = row['topic_name']
        keywords = row['keywords']
        avg = row['compound_mean']
        median = row['compound_median']
        posts = int(row['compound_count'])
        pos = row['positive_mean']
        neg = row['negative_mean']

        print(f"\n{idx}. {topic_name}")
        print(f"   Keywords: {keywords}")
```

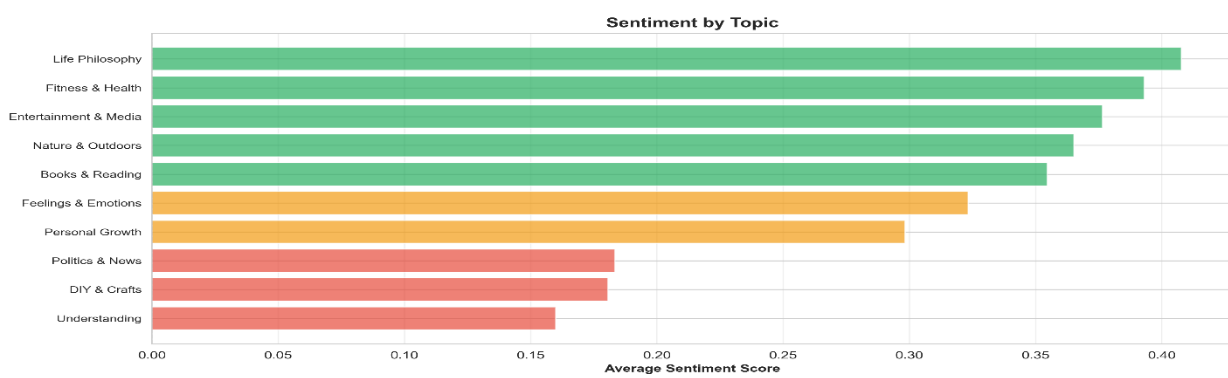
```
        print(f"   Average compound: {avg:.4f} | Median: {median:.4f}")
        print(f"   Number of posts: {posts}")
        print(f"   Positive proportion: {pos:.3f} | Negative proportion:
{neg:.3f}")

    print("\nTOP 3 MOST NEGATIVE TOPICS (Actually 'Least Positive')")

    top_negative = topic_sentiment.tail(3).iloc[::-1]  # Reverse to show most
negative first
    for idx, (_, row) in enumerate(top_negative.iterrows(), 1):
        topic_name = row['topic_name']
        keywords = row['keywords']
        avg = row['compound_mean']
        median = row['compound_median']
        posts = int(row['compound_count'])
        pos = row['positive_mean']
        neg = row['negative_mean']

        print(f"\n{idx}. {topic_name}")
        print(f"   Keywords: {keywords}")
        print(f"   Average compound: {avg:.4f} | Median: {median:.4f}")
        print(f"   Number of posts: {posts}")
        print(f"   Positive proportion: {pos:.3f} | Negative proportion:
{neg:.3f}")

    return topic_sentiment

"""
I create this visualization function to easily see sentiment distribution
The first plot shows average sentiment by topic
The second plot compares sentiment distribution between posts and comments
"""
def visualize_sentiment_analysis(posts_df, comments_df, topic_sentiment):
    sns.set_style("whitegrid")
    fig, axes = plt.subplots(2, 1, figsize=(12, 10))

    # Sentiment by Topic Plot
    plot_data = topic_sentiment.sort_values('compound_mean', ascending=True)
    colors = ['#e74c3c' if s < 0.2 else '#f39c12' if s < 0.35 else '#27ae60'
              for s in plot_data['compound_mean']]

    axes[0].barh(range(len(plot_data)), plot_data['compound_mean'],
color=colors, alpha=0.7)
    axes[0].set_yticks(range(len(plot_data)))
```

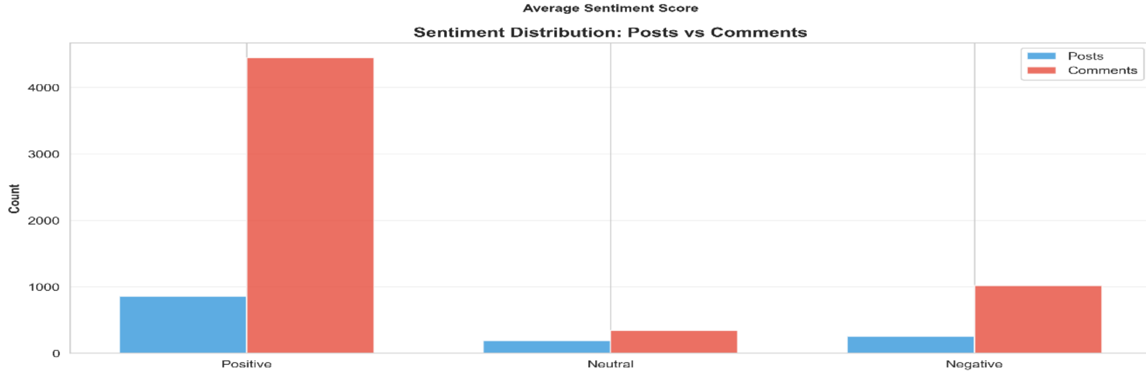
```
    axes[0].set_yticklabels([f"{row['topic_name'][:25]}" for _, row in
plot_data.iterrows()], fontsize=9)
    axes[0].set_xlabel('Average Sentiment Score', fontweight='bold')
    axes[0].set_title('Sentiment by Topic', fontsize=13, fontweight='bold')
    axes[0].axvline(x=0, color='black', linestyle='--', linewidth=1)
    axes[0].grid(axis='x', alpha=0.3)

    # Posts vs Comments Distribution Plot
    categories = ['Positive', 'Neutral', 'Negative']
    posts_vals = [posts_df['sentiment_category'].value_counts().get(c, 0) for c
in categories]
    comments_vals = [comments_df['sentiment_category'].value_counts().get(c, 0)
for c in categories]

    x = np.arange(len(categories))
    width = 0.35
    axes[1].bar(x - width/2, posts_vals, width, label='Posts', color='#3498db',
alpha=0.8)
    axes[1].bar(x + width/2, comments_vals, width, label='Comments',
color='#e74c3c', alpha=0.8)
    axes[1].set_xticks(x)
    axes[1].set_xticklabels(categories)
    axes[1].set_ylabel('Count', fontweight='bold')
    axes[1].set_title('Sentiment Distribution: Posts vs Comments', fontsize=13,
fontweight='bold')
    axes[1].legend()
    axes[1].grid(axis='y', alpha=0.3)

    plt.tight_layout()
    plt.savefig('sentiment_visualization.png', dpi=300, bbox_inches='tight')
    print(f"\nVisualization saved as 'sentiment_visualization.png'")
    plt.close()

def main():
    # Step 1: Analyze sentiment of posts
    posts_df = analyze_posts_sentiment()

    # Step 2: Analyze sentiment of comments
    comments_df = analyze_comments_sentiment()

    # Step 3: Display overall platform sentiment
    display_overall_sentiment(posts_df, comments_df)

    # Step 4: Assign topics to posts (using LDA from Exercise 4.1)
```

```
    posts_df, topic_labels, topic_keywords = assign_topics_to_posts(posts_df)

    # Step 5: Analyze sentiment variation across topics
    topic_sentiment = analyze_sentiment_by_topic(posts_df, topic_labels,
topic_keywords)

    # Step 6: Visualize sentiment analysis
    visualize_sentiment_analysis(posts_df, comments_df, topic_sentiment)

if __name__ == "__main__":
    try:
        main()
    except Exception as e:
        print(f"Error: {e}")
        import traceback
        traceback.print_exc()
    finally:
        conn.close()

"""
Analyzing 1303 posts...
Analyzing 5804 comments...

--- POSTS SENTIMENT ---
Average compound score: 0.3053
Median compound score: 0.4404
Std deviation: 0.4780

Sentiment distribution:
  Positive: 857 (65.8%)
  Neutral: 191 (14.7%)
  Negative: 255 (19.6%)

--- COMMENTS SENTIMENT ---
Average compound score: 0.4324
Median compound score: 0.5983
Std deviation: 0.4836

Sentiment distribution:
  Positive: 4446 (76.6%)
  Neutral: 339 (5.8%)
  Negative: 1019 (17.6%)

--- OVERALL PLATFORM TONE ---
```

```
Average compound score (all content): 0.4091
Overall platform tone: POSITIVE

The platform has a slightly positive tone
--- Assigning topics to posts using LDA model ---

Preprocessing posts...
Loading LDA model and dictionary from disk...
Creating corpus using loaded dictionary...
Assigning topics to posts...
Successfully assigned topics to 1303 posts

Sentiment Summary by Topic (sorted by average compound score):
Rank   Topic Name                         Avg Score    Posts    Pos%     Neg%
--------------------------------------------------------------------------------
-
1      Life Philosophy                    0.4078       117      19.8     3.4
2      Fitness & Health                   0.3930       154      19.1     3.8
3      Entertainment & Media              0.3766       101      20.7     4.5
4      Nature & Outdoors                  0.3652       169      18.2     3.9
5      Books & Reading                    0.3546       119      19.3     5.0
6      Feelings & Emotions                0.3234       146      19.1     5.6
7      Personal Growth                    0.2984       110      20.0     6.6
8      Politics & News                    0.1833       145      16.1     8.1
9      DIY & Crafts                       0.1806       112      15.3     8.3
10     Understanding                      0.1600       130      16.1     10.3

TOP 3 MOST POSITIVE TOPICS

1. Life Philosophy
   Keywords: life, good, need
   Average compound: 0.4078 | Median: 0.5719
   Number of posts: 117
   Positive proportion: 0.198 | Negative proportion: 0.034

2. Fitness & Health
   Keywords: see, health, mental
   Average compound: 0.3930 | Median: 0.5106
   Number of posts: 154
   Positive proportion: 0.191 | Negative proportion: 0.038

3. Entertainment & Media
   Keywords: cant, coffee, best
   Average compound: 0.3766 | Median: 0.5145
```

```
   Number of posts: 101
   Positive proportion: 0.207 | Negative proportion: 0.045

TOP 3 MOST NEGATIVE TOPICS (Actually 'Least Positive')

1. Understanding
   Keywords: kid, could, knew
   Average compound: 0.1600 | Median: 0.1755
   Number of posts: 130
   Positive proportion: 0.161 | Negative proportion: 0.103

2. DIY & Crafts
   Keywords: diy, project, feel
   Average compound: 0.1806 | Median: 0.2490
   Number of posts: 112
   Positive proportion: 0.153 | Negative proportion: 0.083

3. Politics & News
   Keywords: another, anyone, else
   Average compound: 0.1833 | Median: 0.2023
   Number of posts: 145
   Positive proportion: 0.161 | Negative proportion: 0.081

Visualization saved as 'sentiment_visualization.png'
"""
```





Exercise 4.3 Learning from others' mistakes: Find two social platforms similar to Mini Social that have been under fire for an engineering, design or operation error that severely affected a large group of users. Describe how we can learn from their mistakes and draft up a plan about how Mini Social can be improved learning from their mistakes. You do not need to write code in this exercise unless your plan includes a specific change to an algorithm or function. (5 points)

Building a resilient social platform requires learning from the failures of predecessors. Two of the categories that I found most severe errors often fall into are design/operational failures that break community trust and engineering failures that lead to catastrophic technical collapse.

A critical failure in platform design occurred at Reddit in 2023 because of its decision to begin charging for access to API, which had a severe design consequence. It effectively terminated popular third-party applications and affected many users. For instance, volunteer moderators, who relied on these apps for essential tools to manage their communities, were critically impacted. The community perceived this as a betrayal, leading to a massive, coordinated blackout where thousands of subreddits (forums within Reddit) with hundreds of millions of subscribers collectively went offline for days.

The second failure that I want to mention is the catastrophic operational failure occurred at Meta in October 2021. During a routine maintenance task on their global backbone network, a single erroneous command was executed. One of their audit tools contained a bug, was not terminated, and this command promptly withdrew all Border Gateway Protocol routes for Meta's data centers, effectively making Facebook, Instagram, and WhatsApp vanish from the internet for over six hours. Moreover, Meta's infrastructure was so centralized, internal communications, emails, and critical physical access key card systems for the data center also went down. Engineers lost access to the hardware needed to resolve the situation.

Discussion and Improvement Plan for Mini Social

Learning from Reddit, my first advice is to have transparent and inclusive decision-making in Mini Social. This means we should treat core users as partners, not just metrics, by actively consulting them with surveys or polls before making major changes, especially to core features or terms of service. We could also establish permanent feedback channels to ensure community voices are heard and integrated into the development cycle. Ultimately, this will help build community trust and engagement. Honesty is a key factor here. For example, we should be transparent after any significant failures, detailing what went wrong and what we are doing to fix it. We could also go beyond just listening and find ways to formally recognize and reward high-value community contributions that make Mini Social a worthwhile platform.

In terms of technical resilience, my plan is to create a robust incident response plan. We should operate on the assumption that a major failure will happen, so we must develop and test response plans for both technical and community crises. A lesson I have learned from Meta, which I think is very important, is decoupling all critical recovery tools from the main production infrastructure. We can make the admin panels, internal chat, and physical security systems run on completely separate, independent networks. With this approach, we can minimize the risk of Mini Social going down completely.

As seen in these cases, and there could be a lot more out there, a platform can be brought down by a bad line of code or by a bad business decision that breaks the trust of its community. It is best for Mini Social's strategy to prioritize both.

References

- Janardhan, S. (2021, October 5). More details about the October 4 outage. Engineering at Meta. https://engineering.fb.com/2021/10/05/networking-traffic/outage-details/

- Krebs, B. (2021, October 4). What Happened to Facebook, Instagram, & WhatsApp? Krebs on Security. https://krebsonsecurity.com/2021/10/what-happened-to-facebook-instagram-whatsapp/

- Paul, K. (2023, December 30). How social media's biggest user protest rocked Reddit. The Guardian. https://www.theguardian.com/technology/2023/dec/30/reddit-moderator-protest-communities-social-media

Exercise 4.4 Design and implement a new social feature in Mini Social. For example, a user reputation scoring system, a reporting system, a feature to find related content to a post, new post modalities such as polls or reposts. Your change must include a UI improvement or addition. Do not implement non-social, technical features, such as resource optimization, security improvements or style changes. Document the design and implementation process of your addition here. You must also demonstrate a fully functional feature in a maximum 2-minute video recording uploaded to Moodle. (5 points)

For this task, I decided to implement a reporting system based on the findings that I made in task 4.3. The need for a place where people can express their thought could help in making the Mini Social better.

Design overview

I extended Mini Social's features to support two report types, which are post-bound reports and global reports. The global report lets users report platform-level issues and the post-bound one lets user report problem related to a specific post. To manage these report properly, I also add a report tab in the admin page. The admin can review the report, filter ones that are not yet checked, and take actions based on that.

Implementation

1. Data model
   First, I create the report table with columns **id**, **post_id** (nullable for global report), **reported_id**, **reason**, **status**, and **created_at**. Report table is not automatically created by the app, so we have to run the provided script '**create_reports_table.py**' in the project root to create the table manually before using admin

2. Server changes
   The POST /report endpoint was implemented. It will check for the post_id, if it is existing, there will be a notification to user to prevent duplicate open reports by the same reporter for the same post. If it is absent, then it will be a global report. The admin dashboard now supports a filter for post that are reviewed or not. For this, I decided to implement a server-side filter, which will use a query to get all posts instead of doing that in the UI because it could conflict with the pagination feature. The other actions for admin are mark a post reviewed, dismiss the report, or delete the post that they found bad.

3. User interface changes
   I added a global report button in the navigation bar, which will open a report modal on pressed. The global modal does not include a post id field as I mentioned earlier and offers a set of common reasons (bug report, feature request, performance issue, UI/UX problem). Per-post reporting modals remain the same, but with a different set of reasons. The admin has a new Reports tab that shows all reports and a button to filter out the reviewed ones.

Commit URLs:
- feat: https://github.com/HarryxDD/mini_social_exercise/commit/0b08e0387efb89a7d65cf2d077569b855b5b693f
- fix: https://github.com/HarryxDD/mini_social_exercise/commit/2b4cd18220c2c32ad96664e3e4d79728df41bc9d