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, 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

Mohammadhosein Mohammadirad

Student ID

2409311

Student Email

mmohamma24@student.oulu.fi

GitHub Repository URL

https://github.com/mhmohammadirad/SocialComputing

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.

HW1: No use of AI yet
HW2: I faced some errors because of timestamp format in exercise 2.1, which I couldn't understand. I used GPT to find and fix them.

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)

I used SQLite (by Alexcvzz) and SQLite viewer (by Florian Klampfer) extensions in VSCode to load and view the database. I know that using sqlite3 library in python is also an approach, but I used this extension and it is much easier to work with SQLite database. It loads the database so you can see the tables, the table definitions (DDL), and the column data types, and even data in each table.

There are 6 tables in the database. I will go through each of these tables, their columns, their purposes, and data types.

comments table (5 columns, 5804 rows): This table contains information about comments, including which user has created this comment for which post and when.

- *id: unique identifier ranging from 1 to 5804 with INTEGER type.*
- *post_id: foreign key to posts table with INTEGER type to show which post this comment is related to.*
- *user_id: foreign key to users table with INTEGER type to show which user has created this comment*
- *content: the content of the comment with TEXT type e. g.: Don't miss this! Everyone is talking about it ? viralbuzz.news*
- *created_at: creation date and time of the comment with TIMESTAMP type. It is in YYYY-MM-DD hh:mm:ss format.*

follows table (2 columns, 7225 rows): This table contains information about following relationships, showing which user follows which other user

- *follower_id: foreign key to users table with INTEGER type to show which user is following someone (follower)*
- *followed_id: foreign key to users table with INTEGER type to show which user is being followed by someone (following)*

posts table (4 columns, 1303 rows): This table contains information about posts published on this platform, including the content of the post, its creator, and time of creation.

- *id: unique identifier ranging from 1718 to 3020 with INTEGER type.*
- *user_id: foreign key to users table with INTEGER type to show which user has created this post*
- *content: the content of the post with TEXT type e. g.: Revolutionary idea! #fashionblogger #instafashion #model #runway #hairstyle #ootd #trend*
- *created_at: creation date and time of the post with TIMESTAMP type. It is in YYYY-MM-DD hh:mm:ss format.*

reactions table (4 columns, 8276 rows): This table contains information about reactions to the posts, including which user has reacted to which post, how was the reaction, and when.

- *id: unique identifier ranging from 1 to 8286 with INTEGER type.*
- *post_id: foreign key to posts table with INTEGER type to show which post this reaction is related to.*
- *user_id: foreign key to users table with INTEGER type to show which user has reacted to the post*
- *reaction_type: reaction to that post with TEXT type, but it is somehow an ENUM because it has only 6 different values: like, wow, love, haha, sad, angry. I used this query to find them sorted based on their frequency:*

```sql
SELECT reaction_type, COUNT(1) AS count
FROM reactions
GROUP BY reaction_type
ORDER BY 2 DESC;
```

sqlite_sequence (2 columns, 3 rows): This table contains the sequences used for unique identifiers in our table. There are three entries in this table, each of them showing the last used id in a sequence (for the related table). SQLite uses these sequences for autoincrement id sequences.

- *name: name of the sequence with UNKNOWN type which is the name of the table this sequence is related to e. g. reactions, comments, posts*
- *seq: last used number for that sequence.*

users table (7 columns, 210 rows): This table contains information about users on the platform, including their username, password, location, and birthdate.

- *id: unique identifier ranging from 1 to 533 with INTEGER type.*
- *username: username of the user with VARCHAR(50) type. E. g. traveller_tom*
- *location: location (city) of the user with VARCHAR(100) type. E. g. Berlin, Germany*
- *birthdate: birthdate of the user with DATE type. E. g. 1990-10-12*
- *created_at: date and time that the user account was created with TIMESTAMP type. E. g. 2022-05-17 17:32:48*
- *profile: some info that users mention about themselves (like bio) with TEXT type. E. g. Berlin native, born on 10/12/90. ? Cinephile with a penchant for thought-provoking films and ...*
- *password: password of the user with TEXT type. E. g. wLx6NQR1*

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)

There are 55 users who have never interacted with posts or created any content.

*If we replace COUNT(1) with * we can see the details of these users.*

WHERE clause uses three subqueries:

- *All user ids who have created at least one post from posts table.*
- *All user ids who have reacted to at least one post from reactions table.*
- *All user ids who have written at least one comment (this one is optional, since the exercise is asking for posts and reaction. But posts already cover them, so it doesn't affect the result).*

The main query then selects all users whose IDs are not in any of these subqueries.

```sql
SELECT COUNT(1) AS count
FROM users
WHERE id NOT IN (SELECT DISTINCT user_id FROM posts)
  AND id NOT IN (SELECT DISTINCT user_id FROM reactions)
  AND id NOT IN (SELECT DISTINCT user_id FROM comments);
```

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)

For calculating reactions for each user, first we need to join reactions table with posts table to find the publisher of each post and find how many reactions they have received for each of their posts. We do so in the subquery reacts, so we will know which user, on which post, has received how many reactions.
Then, to find the users with highest reaction, we can have two approaches:
1) If we consider the count of all reactions each user have received (sum of all reactions of their posts) then we have these users as the top 5 users with most reactions: 54, 94, 65, 81, 111
2) but as you can see, these users can have a lot of posts with a few reactions on each of them which at the end, creates a high number of overall received reactions, but in fact, they don't receive much reactions, they just post a lot. So we can have a better approach, average reactions that each user receive for each of their posts. In that case, these users have the highest engagement: 78, 43, 109, 94, 13

To calculate engagement based on reactions, we first need to know how many reactions each user has received on their posts. To do this, we join the reactions table with the posts table in order to connect each reaction to the publisher of the post. In the subquery reacts, we calculate how many reactions each post has received, while keeping track of which user owns that post. This gives us a dataset showing, for each user and each post, the number of reactions received.
From this point, there are two possible approaches to identify the top 5 users with the most engagement:

- *Total reactions approach: We sum up all the reactions that each user has received across all their posts. This way, we find the users who have the largest total number of reactions. Using this method, the top 5 users are: 54, 94, 65, 81, 111.*

- *Average reactions approach: A potential problem with the first method is that a user could have many posts with only a few reactions each. Their total reactions would still be high, but it wouldn't necessarily mean their individual posts are highly engaging. To account for this, we can instead look at the average number of reactions per post for each user. This shows us who consistently receives strong engagement on their posts, regardless of how many they make. Using this method, the top 5 users are: 78, 43, 109, 94, 13.*

The query below calculates both total reactions and average reactions per post. By changing the ORDER BY clause, we can switch between the two approaches:

```
WITH reacts AS (
        SELECT posts.user_id, post_id, COUNT(1) AS count
        FROM reactions JOIN posts ON reactions.post_id = posts.id
        GROUP BY posts.user_id, post_id
    )
SELECT user_id, SUM(count) AS total_reactions,
       COUNT(1) AS total_posts, AVG(count) AS avg_reactions_per_post
FROM reacts
GROUP BY user_id
-- ORDER BY 2 DESC -- Using total_reactions(approach 1)
ORDER BY 4 DESC -- Using avg_reactions_per_post (approach 2)
LIMIT 5;
```

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)

There are 5 users who have posted or commented the same text at least 3 times.
ids: 513, 521, 524, 530, 533

First, we combine all posts and comments into one dataset (all_data).
Then, we group by user_id and content to find cases where the same user repeated the same text 3 or more times (spams).
Finally, we select the ids of those users.

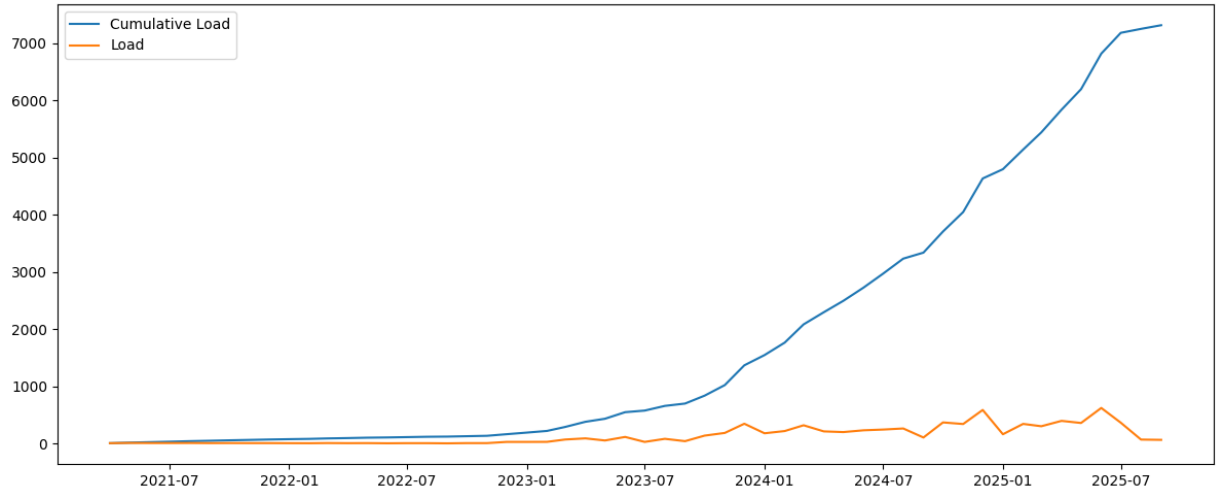
```
WITH all_data AS (
        SELECT user_id, content FROM posts
        UNION ALL
        SELECT user_id, content FROM comments
    ), spams AS (
        SELECT user_id, content, COUNT(1) AS count
        FROM all_data
        GROUP BY user_id, content
        HAVING COUNT(1) >= 3
```

```
    )
SELECT DISTINCT user_id
FROM spams;
```

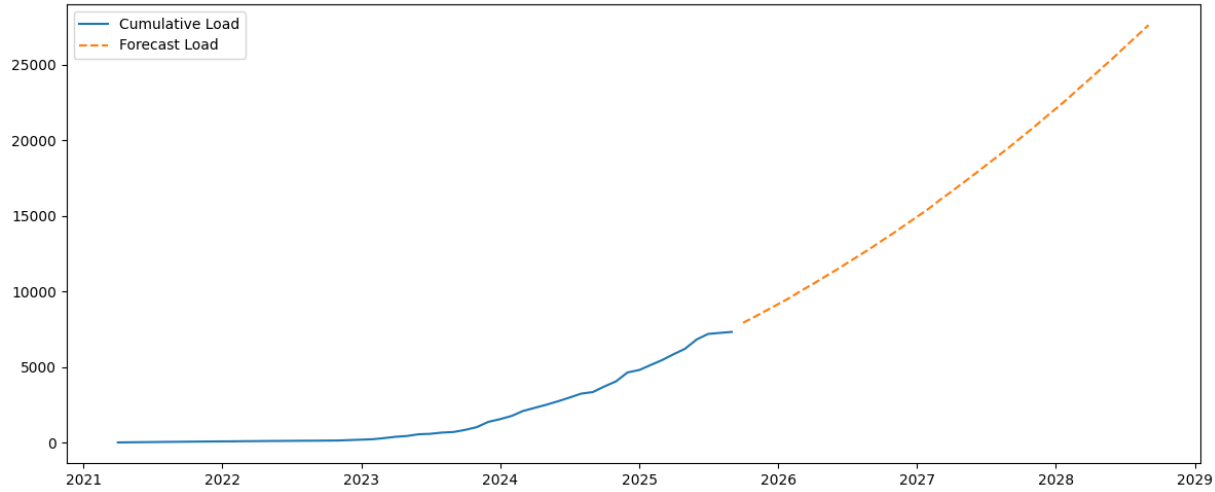
Task 2 (due 29.9.2025 23:59) 15 points

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)

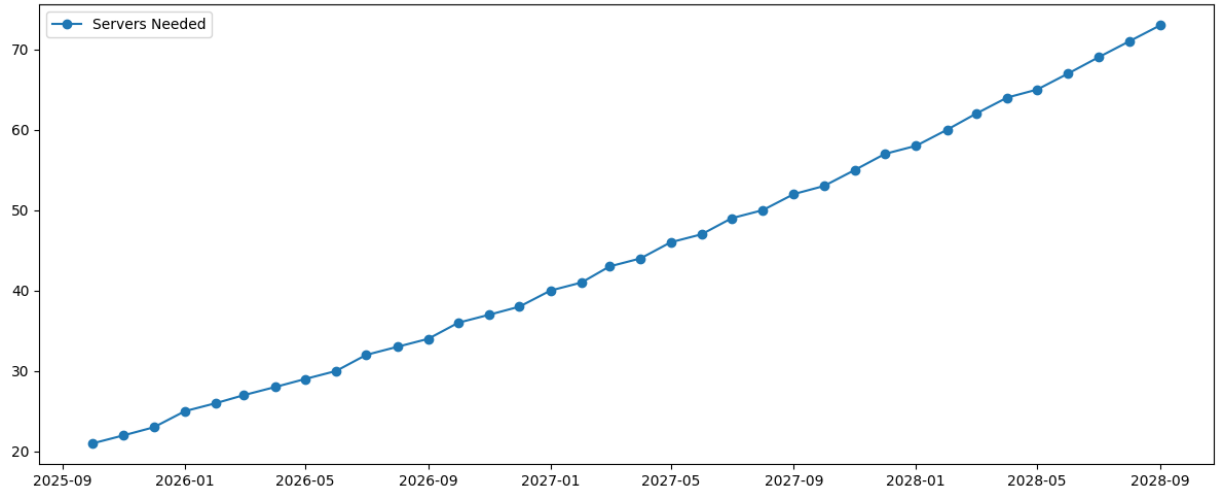
To estimate the future need for servers, we first assessed the current server usage and its trend. I used the posts, comments, and users tables to estimate the load. I couldn't include reactions and follows because these tables don't have a creation timestamp, which is needed for forecasting based on time. We calculated the monthly count of posts, comments, and new users (grouping related tables by created_at column in month format) and defined monthly load as the sum of these counts. This gives an estimate of the load for each month. Initially, I tried to predict the future trend using this monthly load, but it fluctuates a lot and makes it hard to see a clear trend. Instead, I used the cumulative sum of the monthly load.



The cumulative sum of load shows a clear exponential growth trend. To model this, I applied a polynomial transformation and then used linear regression to predict the load for the next 36 months (3 years). The last month in our data is 2025-09, which we consider as today. Our predictions cover until 2028-09.



Now that we have the predicted load, we can estimate the number of servers needed. In the last month, the cumulative load is 7313, which is handled by 16 servers. This gives a load-per-server coefficient. For each predicted month, we divide the predicted load by this coefficient and then multiply by 1.2 to include 20% redundancy. By the end of 36 months, we estimate that we will need 73 servers to handle the load.



In this analysis, we considered load as the total amount of data in our database. Other approaches are also possible, such as using monthly load instead of cumulative load, or weighting posts, comments, and new users differently, because for example, a new user might generate more overall activity than a single new post.

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)

I used two subqueries to calculate engagement for each post.

- *In the first subquery (comments_count), I counted the number of comments for each post.*
- *In the second subquery (reactions_count), I counted the number of reactions for each post.*

Then, I joined these two subqueries using a FULL JOIN so that we can have both counts for each post. After checking the data, I saw that comments are always present, and only a few posts have NULL for reactions count. Therefore, I used the post_id from comments_count because it is never null. For posts where reactions are null, I used the COALESCE function to replace NULL with 0. I then defined a total virality score as:
total = count of comments + count of reactions
We could also assign weights if we want, for example, valuing a comment more than a reaction, but here I kept it simple and just summed them.
Using this method, the top 3 most viral posts are: 2351, 2813, and 2195.

```sql
WITH comments_count AS (
        SELECT post_id, COUNT(1) AS cnt
        FROM comments
        GROUP BY 1
    ), reactions_count AS (
        SELECT post_id, COUNT(1) AS cnt
        FROM reactions
        GROUP BY 1
    )
SELECT cc.post_id, cc.cnt AS comments, rc.cnt AS reactions,
       cc.cnt + COALESCE(rc.cnt, 0) AS total
FROM comments_count cc
    FULL JOIN reactions_count rc ON cc.post_id = rc.post_id
ORDER BY 4 DESC
LIMIT 3;
```

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)

Since reactions don't have timestamps, I only used comments as the measure of engagement. In the subquery (cte), I joined the posts table with the comments table. For each post, I calculated the earliest comment (MIN(created_at)) and the latest comment (MAX(created_at)). After that, I used the JULIANDAY function to find the difference between the post creation time and these first and last comment times.

JULIANDAY returns the date as a number, so by subtracting two dates and multiplying by 24, we get the difference in hours.
Finally, I calculated the average of these differences across all posts for both the first and the last comment.
The results show that, on average:

- *A post receives its first comment about 45.2 hours (about 1.9 days) after being published. This means posts usually start to get noticed within the first two days*
- *A post receives its last comment about 139.0 hours (about 5.8 days) after being published. This means posts usually continues to attract engagement for almost a week.*

```
WITH cte AS (
        SELECT post_id, posts.created_at AS post_created,
            MIN(comments.created_at) AS first_comment,
            MAX(comments.created_at) AS last_comment,
            24 * (JULIANDAY(MIN(comments.created_at)) - JULIANDAY(posts.created_at))
AS first_comment_diff,
            24 * (JULIANDAY(MAX(comments.created_at)) - JULIANDAY(posts.created_at))
AS last_comment_diff,
            COUNT(1) AS cnt
        FROM posts
            JOIN comments ON posts.id = comments.post_id
        GROUP BY 1, 2
)
SELECT AVG(first_comment_diff) AS average_hours_first_comment,
       AVG(last_comment_diff) AS average_hours_last_comment
FROM cte;
```

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)

I defined engagement between two users as the number of times one user comments on or reacts to the other user's posts. To measure this, I considered both directions (user A engaging with user B and user B engaging with user A), and then counted the total engagements between each pair.
In the first subquery, I counted the number of comments a user made on another user's posts.
In the second subquery, I did the same for reactions.
I combined these two using UNION ALL so that all engagements are considered together.
Engagement is mutual, so, I used a CASE WHEN to always order the pair of users in a consistent way (the smaller user ID is user1 and the larger is user2). This way, (user A, user B) and (user B, user A) are treated as the same pair.
Finally, I summed up the total engagements between each pair and selected the top 3.
The result shows the top 3 user pairs with the highest engagement:

- *Users 38 and 88 with 16 engagements*
- *Users 9 and 51 with 13 engagements*
- *Users 13 and 54 with 13 engagements*

```sql
WITH comment_engagement AS (
    SELECT posts.user_id AS author_id,
           comments.user_id AS engager_id,
           COUNT(1) AS cnt
    FROM comments
        JOIN posts ON comments.post_id = posts.id
    GROUP BY 1, 2
),
reaction_engagement AS (
    SELECT posts.user_id AS author_id,
           reactions.user_id AS engager_id,
           COUNT(1) AS cnt
    FROM reactions
    JOIN posts ON reactions.post_id = posts.id
    GROUP BY 1, 2
),
all_engagement AS (
    SELECT author_id, engager_id, cnt FROM comment_engagement
    UNION ALL
    SELECT author_id, engager_id, cnt FROM reaction_engagement
),
pair_engagement AS (
    SELECT CASE WHEN author_id < engager_id THEN author_id ELSE engager_id END AS
user1,
           CASE WHEN author_id < engager_id THEN engager_id ELSE author_id END AS
user2,
           SUM(cnt) AS total_engagement
    FROM all_engagement
    GROUP BY 1, 2
)
SELECT user1, user2, total_engagement
FROM pair_engagement
ORDER BY 3 DESC
LIMIT 3;
```

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)

The implementation is divided into two stages, following the guidelines from the Rules page. It starts by checking for the most severe violations (Tier 1 and Tier 2).
If any of these appear, the function immediately removes the content and returns a fixed score of 5.0.
Otherwise, the text proceeds to a second stage where milder violations are scored and filtered. In stage 2, Tier 3 words are replaced with asterisks, links are removed, and the score increases accordingly.
Excessive capitalization slightly increases the score but does not modify the text.

To make the system safer, I added fuzzy word detection using the Levenshtein distance. This allows the function to catch intentional or unintentional misspellings of offensive words (for example, "stuupiid" or "haate") by comparing word similarity rather than requiring an exact match. Words with at least 80% similarity to a banned term are censored and scored as violations.

```python
# Exercise 3.1
import re
import Levenshtein

def moderate_content(text):
    tier1_words = ["terrorism", "kill", "murder"]
    tier2_phrases = ["make money fast", "buy followers", "click this link"]
    tier3_words = ["stupid", "idiot", "hate", "dumb"]

    censored_text = text.lower()

    # Stage 1.1: severe checks
    for word in tier1_words:
        if re.search(rf"\b{word}\b", censored_text):
            return "[content removed due to severe violation]", 5.0
    for phrase in tier2_phrases:
        if re.search(re.escape(phrase), censored_text):
            return "[content removed due to spam/scam policy]", 5.0
```

```
    # Stage 1.2: scored filtering
    score = 0.0

    # Tier 3 + fuzzy check
    words_in_text = re.findall(r"\b\w+\b", censored_text)
    for word in words_in_text:
        for bad_word in tier3_words:
            similarity = 1 - Levenshtein.distance(word, bad_word) / max(len(word),
len(bad_word))
            if similarity >= 0.8:
                censored_text = re.sub(rf"\b{word}\b", "*" * len(word),
censored_text, flags=re.IGNORECASE)
                score += 2.0
                break

    # Link detection
    links = re.findall(r"(https?://\S+|www\.\S+)", censored_text)
    if links:
        censored_text = re.sub(r"(https?://\S+|www\.\S+)", "[link removed]",
censored_text)
        score += 2.0 * len(links)

    # Excessive capitalization
    letters = [c for c in text if c.isalpha()]
    if len(letters) > 15:
        upper_ratio = sum(1 for c in letters if c.isupper()) / len(letters)
        if upper_ratio > 0.7:
            score += 0.5

    return censored_text, min(score, 5.0)

text = "MAKE MONEY FAST!! This is stupiid: https://spam.com"
print(moderate_content(text))
```

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)

user_risk_analysis function calculates one user's score using the content moderation results and the rules for risk scoring. It considers the risk in their profile introduction, posts, and comments. The score is then adjusted by account age and newer users are treated as more risky. Finally, score is capped at 5.0.

*As an additional improvement, a reaction-based adjustment is added. The proportion of negative reactions ("angry" and "sad") is added to the score. This gives a smooth risk evaluation, meaning users who often respond negatively will gradually accumulate higher risk. ** I used the last day from posts, comments, and users tables as today.*

```
import sqlite3
from datetime import datetime

def get_last_date(cur):
    cur.execute("""
        SELECT MAX(latest_date) FROM (
            SELECT MAX(created_at) AS latest_date FROM users
            UNION
            SELECT MAX(created_at) AS latest_date FROM posts
            UNION
            SELECT MAX(created_at) AS latest_date FROM comments
        )
    """)
    result = cur.fetchone()[0]
    return datetime.strptime(result, "%Y-%m-%d %H:%M:%S")


def user_risk_analysis(cur, user_id, today):

    cur.execute("SELECT profile, created_at FROM users WHERE id = ?", (user_id,))
    row = cur.fetchone()
    if not row:
        return None
    profile, created_at = row

    account_age_days = (today - datetime.strptime(created_at, "%Y-%m-%d
%H:%M:%S")).days

    profile_score = moderate_content(profile)[1] if profile else 0.0

    cur.execute("SELECT content FROM posts WHERE user_id = ?", (user_id,))
    posts = [row[0] for row in cur.fetchall()]
    post_scores = [moderate_content(p)[1] for p in posts] if posts else []
    avg_post_score = sum(post_scores) / len(post_scores) if post_scores else 0.0

    cur.execute("SELECT content FROM comments WHERE user_id = ?", (user_id,))
    comments = [row[0] for row in cur.fetchall()]
    comment_scores = [moderate_content(c)[1] for c in comments] if comments else []
    avg_comment_score = sum(comment_scores) / len(comment_scores) if comment_scores
else 0.0
```

```
    content_risk_score = (profile_score * 1) + (avg_post_score * 3) +
(avg_comment_score * 1)

    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

    cur.execute("""SELECT reaction_type, COUNT(1) FROM reactions WHERE user_id = ?
GROUP BY reaction_type""", (user_id,))
    reactions = dict(cur.fetchall())
    neg_reacts = reactions.get("angry", 0) + reactions.get("sad", 0)
    total_reacts = sum(reactions.values()) if reactions else 0

    if total_reacts > 0:
        negativity_ratio = neg_reacts / total_reacts
        user_risk_score += negativity_ratio

    return min(user_risk_score, 5.0)



conn = sqlite3.connect("/content/database.sqlite")
cur = conn.cursor()
cur.execute("SELECT id, username FROM users")
users = cur.fetchall()

today = get_last_date(cur)

results = []
for user_id, username in users:
    score = user_risk_analysis(cur, user_id, today)
    if score is not None:
        results.append((user_id, username, score))

conn.close()

top5 = sorted(results, key=lambda x: x[2], reverse=True)[:5]
for id, user, score in top5:
    print(f"{id} - {user}: {score}")
```

According to this function, these 5 users have the highest risk:

```
19 - PolarBear: 1.301449275362319
```

```
26 - GoldenEagle: 0.9738095238095238
118 - rockstar_ryan: 0.8484848484848484
22 - NightOwl: 0.8156028368794326
58 - EmeraldCity: 0.7346153846153847
```

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)

In our algorithm we should consider:

- *User's positive reactions (likes, loves, wow): recommend similar posts to those the user reacted positively to.*
- *Followed users' posts: recommend posts created by people the user follows, as they are more likely to be relevant or interesting.*
- *Diverse: avoid recommending the same posts the user has already reacted to.*

We first find posts that the user reacted to positively and extract their keywords (hashtags or frequent words). Then, we search for other posts that contain similar words or come from followed users. Finally, we score and rank these candidate posts, and return the top 5. This strategy includes both personal interests (from positive reactions) and social connections (from followings).

```python
from collections import Counter

def extract_keywords(text):
    if not text:
        return []
    words = re.findall(r"#\w+|\b\w+\b", text.lower())
    words = [w for w in words if len(w) > 2]
    return words


def recommend(cur, user_id):
    cur.execute("""
        SELECT p.id, p.content
        FROM posts p
        JOIN reactions r ON p.id = r.post_id
        WHERE r.user_id = ? AND r.reaction_type IN ('like', 'love', 'wow')
    """, (user_id,))
    liked_posts_data = cur.fetchall()

    liked_posts = [p[0] for p in liked_posts_data]
    keyword_freq = Counter()

    for id, content in liked_posts_data:
```

```
        keyword_freq.update(extract_keywords(content))


    cur.execute("SELECT followed_id FROM follows WHERE follower_id = ?", (user_id,))
    followed_users = [r[0] for r in cur.fetchall()]

    followed_posts = []
    if followed_users:
        cur.execute(f"""
            SELECT id, content
            FROM posts
            WHERE user_id IN ({','.join(['?'] * len(followed_users))})
        """, followed_users)
        followed_posts = cur.fetchall()

    exclude_ids = ",".join(map(str, liked_posts)) if liked_posts else "0"
    cur.execute(f"SELECT id, content FROM posts WHERE id NOT IN ({exclude_ids})")
    all_posts = cur.fetchall()

    scored_posts = []
    for post_id, content in all_posts:
        post_words = extract_keywords(content)
        similarity = sum(keyword_freq[w] for w in post_words if w in keyword_freq)
        scored_posts.append((post_id, similarity))

    for id, content in followed_posts:
        scored_posts.append((id, 1.0))

    scored_posts = sorted(scored_posts, key=lambda x: x[1], reverse=True)
    top_5 = scored_posts[:5]

    recommended = []
    for id, score in top_5:
        cur.execute("SELECT content FROM posts WHERE id = ?", (id,))
        text = cur.fetchone()
        if text:
            recommended.append((id, text[0], round(score, 2)))

    return recommended


conn = sqlite3.connect("/content/database.sqlite")
cur = conn.cursor()

recs = recommend(cur, user_id=42)
for id, text, score in recs:
    print(f"Post {id} (score {score}): {text[:80]}...")
```

For example, for user 42 we have:

```
Post 2527 (score 243): Spent the morning hiking in the mountains.
Nothing like the fresh air to clear t...
Post 1857 (score 222): Spent the day hiking with my family in the
Pentlands. The views were stunning, a...
Post 1782 (score 203): Today I took a stroll through the local park, and
the autumn colors were stunnin...
Post 1984 (score 202): Exploring the streets of Barcelona. The
architecture, the food, the people—it's ...
Post 2639 (score 194): Spent the afternoon capturing portraits in the
park. The light was perfect, and ...
```

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)

Write your answer here…

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 3? 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)

Write your answer here…

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)

Write your answer here…

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)

Write your answer here…