ARATRIK ROY CHOUDHURY

HITWICKET ASSIGNMENT

PRODUCT ANALYST

Retrieve the list of teams that played at least 3 **home** matches of type **'F5'** between timestamp 1583001000 (1st March 2020) and timestamp 1583778600 (9th March 2020), and the number of matches they played

To retrieve the list of teams that played at least 3 home matches of type 'F5' between the given timestamps, you can use the following SQL query:

```
SELECT home_team_id, COUNT(*) AS matches
FROM matches
WHERE type = 'F5'
  AND timestamp BETWEEN 1583001000 AND 1583778600
GROUP BY home_team_id
HAVING COUNT(*) >= 3;
```

Explanation

1. **FROM matches**: We are querying the data from the matches table.
2. **WHERE type = 'F5'**: This condition filters the matches to only those of type 'F5'.
3. **AND timestamp BETWEEN 1583001000 AND 1583778600**: This further filters the matches to only those that occurred between 1st March 2020 (timestamp 1583001000) and 9th March 2020 (timestamp 1583778600).
4. **GROUP BY home_team_id**: We group the results by the home_team_id to aggregate matches played by each team.
5. **COUNT(*)**: Counts the number of matches each home_team_id has played within the specified conditions.
6. **HAVING COUNT(*) >= 3**: This filters the groups to only include teams that played at least 3 matches.

Output

The result will be a list of home_team_id and the corresponding number of matches (as matches) for teams that meet the criteria. This output gives you the teams that played at least 3 home matches of type 'F5' within the specified date range.

Find out all unique users who have made a successful payment between timestamp 1583001000 (1st March 2020) and timestamp 1583778600 (9th March 2020)

To find all unique users who have made a successful payment between the specified timestamps, you can use the following SQL query. This assumes that you have a payments table with a user_id column and a status column indicating whether a payment was successful.

```
SELECT DISTINCT p.user_id

FROM payments p

JOIN users u ON p.user_id = u.user_id

WHERE p.timestamp BETWEEN 1583001000 AND 1583778600

  AND p.status = 'successful';
```

Explanation

1. **FROM payments p**: We are querying the data from the payments table, using an alias p.
2. **JOIN users u ON p.user_id = u.user_id**: We join the payments table with the users table on the user_id column to ensure we are working with valid user data.
3. **WHERE p.timestamp BETWEEN 1583001000 AND 1583778600**: This filters the payments to only those made between 1st March 2020 (timestamp 1583001000) and 9th March 2020 (timestamp 1583778600).
4. **AND p.status = 'successful'**: This ensures that we are only considering payments that were successful.
5. **SELECT DISTINCT p.user_id**: The DISTINCT keyword ensures that each user_id is listed only once in the result, even if a user made multiple successful payments within the specified timeframe.

Output

The result will be a list of unique user_ids who made successful payments between the given timestamps. This output ensures that each user who met the criteria is represented only once.

Find out all users who have **not completed a single payment** between timestamps 1366137000 (17th April 2013) and 1397673000 (17th April 2014) and were still active after 1483209000 (1st Jan 2017)

To find all users who have not completed a single payment between the specified timestamps (17th April 2013 and 17th April 2014) and were still active after 1st January 2017, you can use the following SQL query. This assumes that you have a payments table and a users table as described.

```
SELECT u.user_id

FROM users u

LEFT JOIN payments p

ON u.user_id = p.user_id

  AND p.timestamp BETWEEN 1366137000 AND 1397673000

WHERE p.id IS NULL

  AND u.last_active_at > 1483209000;
```

Explanation

1. **FROM users u**: We start by selecting all users from the users table using the alias u.
2. **LEFT JOIN payments p ON u.user_id = p.user_id AND p.timestamp BETWEEN 1366137000 AND 1397673000**: We perform a left join between the users table and the payments table on the user_id. The join condition includes a filter that only considers payments made between the specified timestamps.
3. **WHERE p.id IS NULL**: This condition ensures that only users who have no matching payments within the specified timeframe are selected. Since it's a left join, if there are no corresponding records in the payments table, p.id will be NULL, indicating no payments were made during that period.
4. **AND u.last_active_at > 1483209000**: This filters the results to include only those users who were still active after 1st January 2017.

Output

The result will be a list of user_ids of users who did not make any payments between 17th April 2013 and 17th April 2014 and were still active after 1st January 2017. This ensures that you identify users

who, despite being active in recent years, were not contributing through payments during the specified earlier period.

Write the syntax to create the following DataFrame

Class	**Gender**	**Values**
4	Female	10
4	Male	3
5	Male	1
1	Female	5

1	Male	7
2	Female	2
5	Female	5
2	Male	10

```
import pandas as pd


# Creating the DataFrame
data = {
    'Class': [4, 4, 5, 1, 1, 2, 5, 2],
    'Gender': ['Female', 'Male', 'Male', 'Female', 'Male', 'Female', 'Female', 'Male'],
    'Values': [10, 3, 1, 5, 7, 2, 5, 10]
}


df = pd.DataFrame(data)


print(df)
```

Write the syntax to sort this DataFrame by Class (ascending order) followed by Values (descending)

```
import pandas as pd

# Creating the DataFrame
data = {
    'Class': [4, 4, 5, 1, 1, 2, 5, 2],
    'Gender': ['Female', 'Male', 'Male', 'Female', 'Male', 'Female', 'Female', 'Male'],
    'Values': [10, 3, 1, 5, 7, 2, 5, 10]
}

df = pd.DataFrame(data)

# Sorting the DataFrame
df_sorted = df.sort_values(by=['Class', 'Values'], ascending=[True, False])

print(df_sorted)
```

Write the syntax to group this DataFrame by Class, Gender and count the number of unique values

```
import pandas as pd

# Creating the DataFrame
data = {
    'Class': [4, 4, 5, 1, 1, 2, 5, 2],
    'Gender': ['Female', 'Male', 'Male', 'Female', 'Male', 'Female', 'Female', 'Male'],
    'Values': [10, 3, 1, 5, 7, 2, 5, 10]
}

df = pd.DataFrame(data)
```

```
# Grouping by Class and Gender, then counting unique Values
result = df.groupby(['Class', 'Gender'])['Values'].nunique().reset_index(name='Unique_Count')

print(result)
```

Explanation

- **Python**:
    - groupby(['Class', 'Gender']) groups the DataFrame by the Class and Gender columns.
    - ['Values'].nunique() calculates the number of unique Values in each group.
    - reset_index(name='Unique_Count') resets the index and renames the count column to Unique_Count.

Write the syntax to visualize the Data in a Scatter Plot in Python (Class on the X-axis, Values in Y), with color differentiation based on Gender.

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

# Creating the DataFrame
data = {
   'Class': [4, 4, 5, 1, 1, 2, 5, 2],
   'Gender': ['Female', 'Male', 'Male', 'Female', 'Male', 'Female', 'Female', 'Male'],
   'Values': [10, 3, 1, 5, 7, 2, 5, 10]
}

df = pd.DataFrame(data)

# Plotting
plt.figure(figsize=(10, 6))
sns.scatterplot(data=df, x='Class', y='Values', hue='Gender', palette='viridis', s=100)

# Adding titles and labels
plt.title('Scatter Plot of Values by Class and Gender')
plt.xlabel('Class')
```

```
plt.ylabel('Values')


# Display the plot
plt.legend(title='Gender')
plt.grid(True)
plt.show()
```

Explanation

1. **Import Libraries**: pandas for data handling, matplotlib.pyplot for plotting, and seaborn for advanced visualization.
2. **Create DataFrame**: Define the data and create the DataFrame df.
3. **Set Up Plot**:
    - plt.figure(figsize=(10, 6)) sets the size of the plot.
    - sns.scatterplot() creates the scatter plot.
        - data=df specifies the DataFrame to use.
        - x='Class' and y='Values' set the X and Y axes, respectively.
        - hue='Gender' differentiates data points by gender with different colors.
        - palette='viridis' specifies the color palette for the plot.
        - s=100 sets the size of the scatter points.
4. **Customize Plot**:
    - plt.title() adds a title to the plot.
    - plt.xlabel() and plt.ylabel() label the X and Y axes.
    - plt.legend() adds a legend with a title for the Gender colors.
    - plt.grid(True) adds a grid for better readability.
5. **Display Plot**: plt.show() renders the plot.

Problem 5

Write the syntax to reshape/summarize the table above, to display the following output

Class	**Male**	**Female**

1	7	5
2	10	2
4	3	10
5	1	5

```
import pandas as pd


# Creating the DataFrame
data = {
    'Class': [4, 4, 5, 1, 1, 2, 5, 2],
    'Gender': ['Female', 'Male', 'Male', 'Female', 'Male', 'Female', 'Female', 'Male'],
    'Values': [10, 3, 1, 5, 7, 2, 5, 10]
}


df = pd.DataFrame(data)


# Pivot table to summarize data
pivot_table = df.pivot_table(index='Class', columns='Gender', values='Values', aggfunc='sum',
fill_value=0)


# Resetting index to convert the pivot table into a regular DataFrame
result = pivot_table.reset_index()


print(result)
```

Problem 6

Use R or Python to plot all of the following graphs at once, in one window, for easier comparison of categories.

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

# Assuming table_to_plot DataFrame is already defined
# Example DataFrame
data = {
    'Category': [1, 1, 2, 2, 3, 3, 4, 4, 5],
    'x_variable': [800, 1200, 900, 1400, 1000, 1600, 1100, 1700, 1200],
    'y_variable': [1000, 1400, 1100, 1500, 1200, 1600, 1300, 1700, 1400]
}
table_to_plot = pd.DataFrame(data)

# Set up the matplotlib figure
plt.figure(figsize=(15, 10))

# Plotting
for category in table_to_plot['Category'].unique():
    subset = table_to_plot[table_to_plot['Category'] == category]
    plt.subplot(3, 3, category)
    sns.scatterplot(data=subset, x='x_variable', y='y_variable')
    plt.title(f'Category {category}')
```

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

Problem 7

Create two additional columns in the dataframe table_to_plot, one corresponding to x_variable and one corresponding to y_variable, such that the values of the two existing columns are assigned labels based on what range they are located in, if we were to divide these columns into 3 equally sized intervals.

```
import pandas as pd

# Example DataFrame
data = {
    'Category': [1, 1, 2, 2, 3, 3, 4, 4, 5],
    'x_variable': [800, 1200, 900, 1400, 1000, 1600, 1100, 1700, 1200],
    'y_variable': [1000, 1400, 1100, 1500, 1200, 1600, 1300, 1700, 1400]
}
table_to_plot = pd.DataFrame(data)

# Creating intervals for x_variable
x_bins = pd.cut(table_to_plot['x_variable'], bins=3, labels=['Low', 'Medium', 'High'])
table_to_plot['x_variable_label'] = x_bins

# Creating intervals for y_variable
y_bins = pd.cut(table_to_plot['y_variable'], bins=3, labels=['Low', 'Medium', 'High'])
table_to_plot['y_variable_label'] = y_bins

print(table_to_plot)
```

Section 3: Problem Solving Approach

New User Onboarding

Analyzing Drop-Offs During the Onboarding Tutorial

To analyze the drop-offs during the onboarding tutorial phase, you should collect the following data points and user feedback:

Specific Data Points and User Feedback

1. **Drop-Off Rate by Tutorial Stage**: Track where users drop off during the tutorial. This could include specific steps or milestones within the tutorial.
2. **Time Spent per Stage**: Measure how long users spend on each stage of the tutorial to identify if certain stages are taking too long or too short.
3. **User Interactions**: Record interactions with the tutorial elements, such as clicks, swipes, or any in-game actions that might indicate confusion or frustration.
4. **Error Logs**: Collect data on any errors or bugs encountered during the tutorial. This includes crashes, incorrect functionality, or unexpected behavior.
5. **User Feedback and Ratings**: Gather direct feedback from users who completed or dropped out of the tutorial. This can include surveys or in-app feedback forms asking about their experience, perceived difficulty, and suggestions for improvement.
6. **User Profiles and Behavior**: Analyze user demographics, previous gaming experience, and behavior patterns to understand if certain user types are more likely to drop off.
7. **Tutorial Performance Metrics**: Track metrics such as completion rates, success rates in tutorial challenges, and progression rates.
8. **Comparison with Retained Users**: Compare the behavior and feedback of users who completed the tutorial with those who dropped off to identify differences.

Top 5 Hypotheses

1. **Hypothesis 1: Tutorial Complexity**
   Users are dropping off because the tutorial is too complex or overwhelming. The steps or instructions may be difficult to understand or follow.
2. **Hypothesis 2: Technical Issues**
   Users encounter technical issues or bugs during the tutorial that prevent them from progressing, leading to frustration and drop-offs.

3. **Hypothesis 3: Engagement and Motivation**
   The tutorial lacks engaging or motivating elements, causing users to lose interest or feel that the game is not worth continuing.

4. **Hypothesis 4: User Expectations**
   The tutorial does not align with users' expectations or preferences. Users may have different play styles or expectations that the tutorial does not address.

5. **Hypothesis 5: Length of the Tutorial**
   The tutorial may be too long or time-consuming, causing users to drop off if they feel it is taking too much time to get to the main gameplay.

Experiment to Validate Top Hypothesis

Hypothesis: Tutorial Complexity

Experiment: A/B Testing with Simplified Tutorial

1. **Create Two Versions of the Tutorial**:
   - **Version A**: Current tutorial as-is.
   - **Version B**: Simplified tutorial with fewer steps, clearer instructions, and more interactive elements.
2. **Randomly Assign New Users**:
   - Randomly assign new users to either Version A or Version B to ensure unbiased results.
3. **Collect and Compare Metrics**:
   - **Drop-Off Rate**: Measure the drop-off rates for both versions.
   - **Completion Rate**: Track the percentage of users who complete the tutorial in each version.
   - **User Feedback**: Collect feedback from users about their experience with both versions of the tutorial.
   - **Time Spent**: Compare the average time spent on the tutorial in each version.
4. **Analyze Results**:
   - Determine if users in the simplified tutorial (Version B) have a lower drop-off rate and higher completion rate compared to the current tutorial (Version A).
   - Assess the feedback to identify if users found the simplified version easier to follow and more engaging.

Reasoning Behind the Proposed Experiment

- **Validation of Hypothesis**: This experiment directly tests whether simplifying the tutorial reduces drop-offs and improves user engagement.
- **Controlled Comparison**: A/B testing provides a controlled environment to compare the effectiveness of the current tutorial versus a simplified version.

- **Actionable Insights**: The results will provide clear insights into whether complexity is a major factor in drop-offs and guide improvements.

Possible Drawbacks

1. **Bias in User Assignment**: If users are not randomly assigned, the results may be skewed by differences in user demographics or behaviors.
2. **Changes in User Behavior**: If the simplified tutorial is significantly different, users might not experience the same game mechanics, which could affect long-term engagement.
3. **External Factors**: Other factors such as concurrent changes in the game or external promotions might influence the results.
4. **Limited Scope**: This experiment focuses on tutorial complexity but may not address other factors like technical issues or engagement, which could also contribute to drop-offs.

By conducting this experiment, you can gain valuable insights into whether simplifying the tutorial resolves drop-off issues and use this data to enhance the onboarding experience for new users.

Feature Improvement

Feature Improvement: In-Game Player Matchmaking

Identified Feature: In-Game Player Matchmaking

Context: In many competitive mobile games, including cricket simulation games like Hitwicket, matchmaking plays a critical role in user satisfaction. If matchmaking is not optimized, players may experience mismatched opponents, which can lead to frustration and decreased engagement.

Data and User Feedback Collection

To establish that the matchmaking feature has room for improvement, collect the following data and feedback:

1. **Match Outcome Data**: Track win/loss ratios, score differences, and match durations to determine if players are frequently facing opponents of mismatched skill levels.
2. **Player Feedback**: Collect direct feedback through surveys or in-game feedback forms asking players about their satisfaction with matchmaking, perceived fairness, and any frustrations experienced.
3. **Matchmaking Queue Times**: Measure the average time players spend waiting for a match to see if longer wait times correlate with dissatisfaction or player drop-offs.
4. **Skill Rating Changes**: Analyze changes in player skill ratings over time to see if the matchmaking system effectively balances skill levels.
5. **Churn Rate**: Monitor the churn rate of players who engage in competitive matches compared to those who do not, to identify if poor matchmaking contributes to player attrition.

Top 3 Suggestions/Ideas for Improvement

1. Implement Dynamic Skill-Based Matchmaking

Description: Use an algorithm that adjusts matchmaking dynamically based on player performance, recent results, and skill ratings to ensure players face opponents of similar skill levels.

Pros:

- **Improved Fairness**: Matches are more balanced, enhancing player satisfaction and engagement.
- **Reduced Frustration**: Players are less likely to feel overmatched or underchallenged.

Cons:

- **Complexity**: Requires advanced algorithms and continuous adjustments, increasing development complexity.
- **Longer Queue Times**: May result in longer wait times for matches if finding equally skilled opponents is challenging.

2. Introduce a Skill Rating Reset Periodically

Description: Periodically reset skill ratings (e.g., every season) to allow players to compete at different levels and prevent skill inflation.

Pros:

- **Revitalizes Gameplay**: Encourages players to re-engage and test their skills anew.
- **Balances Skill Distribution**: Helps in resetting the skill distribution, making matchmaking more effective.

Cons:

- **Player Confusion**: Frequent resets may confuse or frustrate players who prefer consistent skill levels.
- **Potential Imbalance**: Initial matches post-reset may still suffer from imbalances until new ratings stabilize.

3. Add a "Skill Level" Indicator for Opponents

Description: Display an indicator showing the skill level or rank of opponents before starting a match to inform players of the relative difficulty.

Pros:

- **Transparency**: Allows players to gauge the challenge level and make informed decisions about engagement.
- **Reduced Surprises**: Players know what to expect and can adjust their strategy accordingly.

Cons:

- **Potential Pressure**: Could create pressure or anxiety for players facing higher-ranked opponents.
- **Discrepancy Issues**: Might not always accurately reflect the true skill difference due to fluctuating performance.

Experiment and Success Metrics

Experiment: A/B Testing Dynamic Skill-Based Matchmaking

Setup:

1. **Create Two Versions**:
    - **Version A (Control)**: Current matchmaking system.
    - **Version B (Experiment)**: Implement dynamic skill-based matchmaking.
2. **Randomly Assign Players**: Assign new players to either Version A or Version B to ensure unbiased results.
3. **Collect Data**:
    - **Match Outcome Metrics**: Compare win/loss ratios and score differences between versions.
    - **Player Feedback**: Gather feedback on matchmaking fairness and satisfaction.
    - **Queue Times**: Measure and compare average matchmaking wait times.
4. **Analyze Results**:
    - Evaluate if dynamic matchmaking improves match fairness and player satisfaction without significantly increasing queue times.
    - Assess whether players in Version B report higher satisfaction and engagement compared to Version A.

Reasoning Behind the Proposed Experiment

- **Validation of Hypothesis**: Directly tests whether dynamic skill-based matchmaking improves player satisfaction and match fairness.
- **Controlled Comparison**: A/B testing allows a controlled environment to compare the effectiveness of the new system against the current one.
- **Actionable Insights**: Provides clear insights into whether the new matchmaking algorithm offers a tangible improvement.

Possible Drawbacks

1. **Implementation Complexity**: Dynamic matchmaking may introduce technical complexities and require substantial development and maintenance.
2. **Longer Queue Times**: If the system struggles to find well-matched opponents, it could lead to increased wait times, which might deter players.
3. **Balancing Issues**: The new system may need continuous adjustments and fine-tuning to ensure it consistently delivers fair matches.

By conducting this experiment, you can determine if dynamic skill-based matchmaking effectively addresses issues with the current system and enhances the overall player experience.