

Mathball

Congratulations, you are now hired as a sport analytic. Your job is to give prediction for the next Mathball tournament of the Six finest mathematicians in the world, which is a competition between Laplace, Gauss, Newton, Euler, Leibniz, Von-Neumann.

The mathball is just like football where a player a goal is legal only after scoring you answer correctly a math question.

▼ Motivation:

We want to infer a latent parameter - that is the 'strength' of a mathematician based only on their **scoring intensity**, and all we have are their scores and results, we can't accurately measure the 'strength' of a team.

```
[ ] # imports
import arviz as az
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import pymc as pm
import seaborn as sns
import pickle as pkl
```

▼ The input looks like as follows:

```
[ ] group = 24
df_all = pkl.load(open('../content/data_mathball/mathball_data_group_' + str(group) + '.pkl', 'rb'))
df_all.head()
```



The image shows a Jupyter Notebook interface. On the left, there is a sidebar with a file explorer icon and a plus icon. The main area displays a table with 5 columns: home_player, away_player, home_score, away_score, and year. The table has 5 rows of data. To the right of the table, there are two icons: a calendar icon and a bar chart icon.

	home_player	away_player	home_score	away_score	year
0	Laplace	Von-Neumann	27	24	1897
1	Gauss	Euler	28	3	1897
2	Newton	Laplace	17	19	1897
3	Euler	Leibniz	23	15	1897
4	Laplace	Leibniz	31	10	1897

Next steps:

[Generate code with df_all](#)

[View recommended plots](#)

We have home and away team, we have score and year.

What do we want to infer?

We want to infer the latent parameters (every mathematician's strength) that are generating the data we observe (the scorelines).

Moreover, we know that the scorelines are a noisy measurement of team strength, so ideally, we want a model that makes it easy to quantify our uncertainty about the underlying strengths.

What do we want?

We want to quantify our uncertainty

We want to also use this to generate a model

We want the answers as distributions not point estimates

✓ Part 1

We next do some exploratory data analysis. The idea is to get a general sense of what is going on.

Tasks

1. Plot a bar chart of the goal difference (i.e., total scored minus total scored against) per mathematician. Who is the best mathematician and who is the worst according to this plot?
2. Plot the same bar chart for each mathematician as a function of time. Do you see a monotone behavior which implies an improvement or the other way around for one of mathematician?
3. Choose your favorite mathematician, compare the goal difference between home and away games. Is he better in home or away games? Plot a bar chart of the goal difference for both home and away games.
4. Which mathematician has the best median away results? (that is, for each mathematician go over all of his away games, compute his scoring difference (how much he scored minus got scored against) and compute the median value).

▼ Answers to part 1:

1:

```
# Calculate goal difference for home and away players
df_all['home_goal_diff'] = df_all['home_score'] - df_all['away_score']
df_all['away_goal_diff'] = df_all['away_score'] - df_all['home_score']

# Aggregate goal differences for each player
home_goal_diff = df_all.groupby('home_player')['home_goal_diff'].sum().reset_index()
away_goal_diff = df_all.groupby('away_player')['away_goal_diff'].sum().reset_index()

# Merge the home and away goal differences
goal_diff = pd.merge(home_goal_diff, away_goal_diff, left_on='home_player', right_on='away_player', how='outer')
goal_diff = goal_diff.fillna(0)

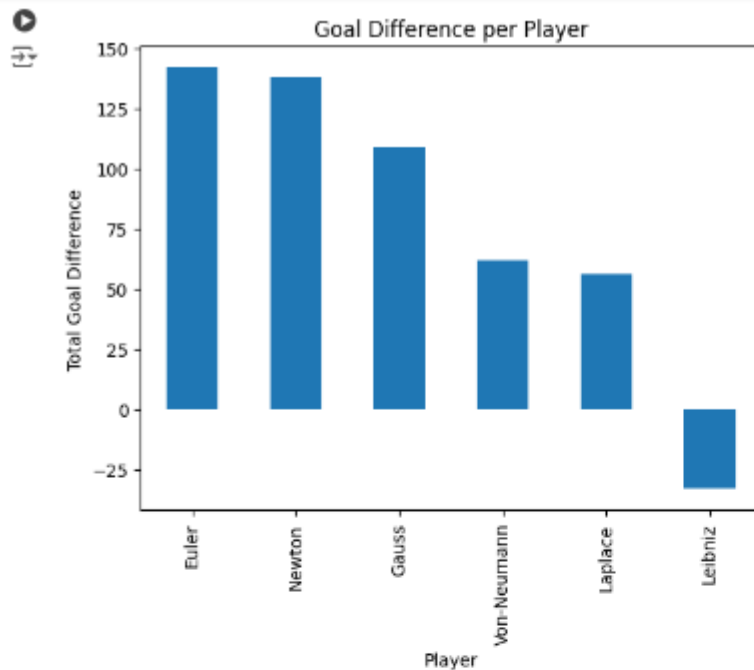
# Calculate the total goal difference
goal_diff['total_goal_diff'] = goal_diff['home_goal_diff'] - goal_diff['away_goal_diff']

# Rename the player column and set it as index
goal_diff['player'] = goal_diff['home_player'].combine_first(goal_diff['away_player'])
goal_diff = goal_diff[['player', 'total_goal_diff']].set_index('player')

# Plotting the bar chart
goal_diff.sort_values(by='total_goal_diff', ascending=False, inplace=True)
goal_diff.plot(kind='bar', legend=False)
plt.title('Goal Difference per Player')
plt.xlabel('Player')
plt.ylabel('Total Goal Difference')
plt.show()

# Determine the best and worst player
best_player = goal_diff['total_goal_diff'].idxmax()
worst_player = goal_diff['total_goal_diff'].idxmin()

print(f"The best mathematician is {best_player} with a goal difference of {goal_diff['total_goal_diff'].max()}.")
print(f"The worst mathematician is {worst_player} with a goal difference of {goal_diff['total_goal_diff'].min()}.")
```



The best mathematician is Euler with a goal difference of 142.
The worst mathematician is Leibniz with a goal difference of -33.

2:

```
# Calculate goal difference for home and away players
df_all['home_goal_diff'] = df_all['home_score'] - df_all['away_score']
df_all['away_goal_diff'] = df_all['away_score'] - df_all['home_score']

# Melt the DataFrame to long format
df_home = df_all[['year', 'home_player', 'home_goal_diff']].rename(columns={'home_player': 'player', 'home_goal_diff': 'goal_diff'})
df_away = df_all[['year', 'away_player', 'away_goal_diff']].rename(columns={'away_player': 'player', 'away_goal_diff': 'goal_diff'})
df_long = pd.concat([df_home, df_away])

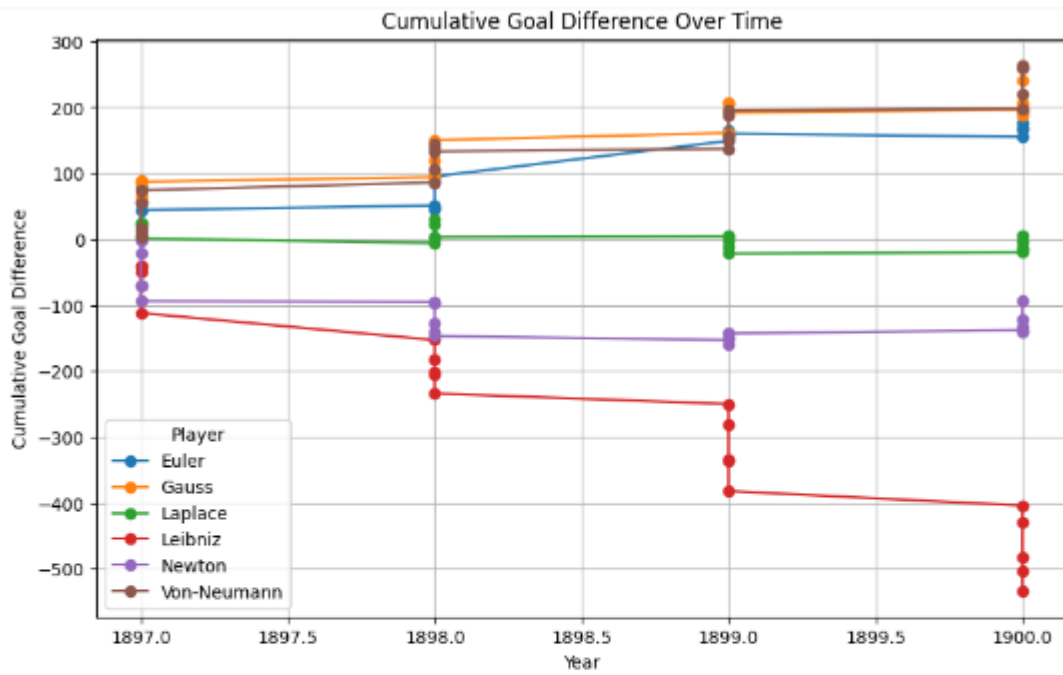
# Sort by player and year for cumulative sum calculation
df_long = df_long.sort_values(by=['player', 'year'])

# Calculate cumulative goal difference
df_long['cumulative_goal_diff'] = df_long.groupby('player')['goal_diff'].cumsum()

# Plotting the cumulative goal difference over time for each player
plt.figure(figsize=(10, 6))
for player in df_long['player'].unique():
    player_data = df_long[df_long['player'] == player]
    plt.plot(player_data['year'], player_data['cumulative_goal_diff'], marker='o', label=player)

plt.title('Cumulative Goal Difference Over Time')
plt.xlabel('Year')
plt.ylabel('Cumulative Goal Difference')
plt.legend(title='Player')
plt.grid(True)
plt.show()
```

[]
[]



According to the graph 'Cumulative Goal Difference Over Time' The following trends can be seen:

1. Gauss and Von-Neumann show a positive monotonic improvement trend.
2. Leibniz shows a negative monotonic improvement trend.
3. Laplace and Newton show a neutral monotonic improvement trend.
4. Euler shows a positive but non-monotonic conservation trend because a jump can be seen between the years 1898-1899.

3:

I gave the right to choose in this section to my mother (Yael's) who did a degree in mathematics and she **chose Euler** because he contributed in many fields, and was amazingly creative (p.s. according to her everyone on the list is like that and Newton is a super genius).

```

# Calculate goal difference for home and away players
df_all['home_goal_diff'] = df_all['home_score'] - df_all['away_score']
df_all['away_goal_diff'] = df_all['away_score'] - df_all['home_score']

# Filter data for Euler
euler_home = df_all[df_all['home_player'] == 'Euler']
euler_away = df_all[df_all['away_player'] == 'Euler']

# Calculate total goal difference for home and away games
euler_home_goal_diff = euler_home['home_goal_diff'].sum()
euler_away_goal_diff = euler_away['away_goal_diff'].sum()

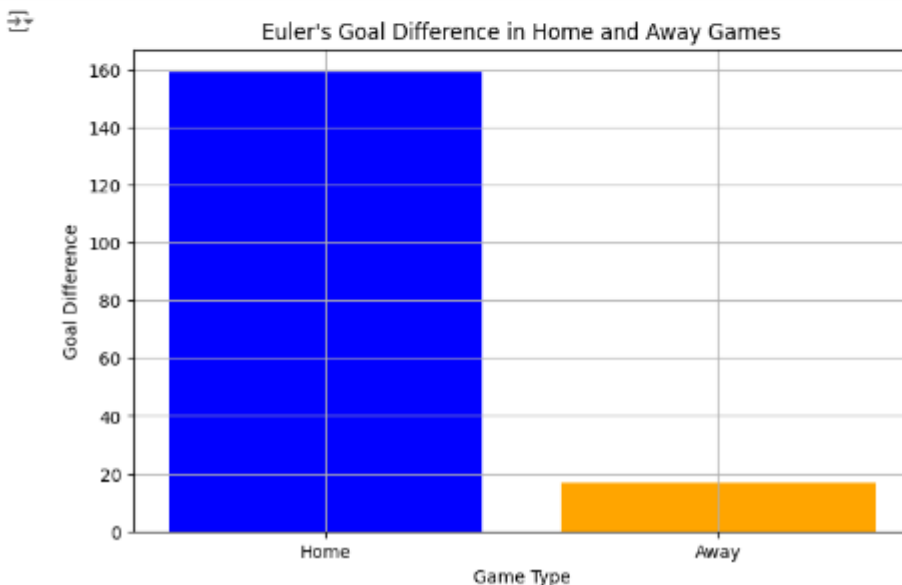
# Create a DataFrame for plotting
euler_goal_diff = pd.DataFrame({
    'Game Type': ['Home', 'Away'],
    'Goal Difference': [euler_home_goal_diff, euler_away_goal_diff]
})

# Plotting the bar chart
plt.figure(figsize=(8, 5))
plt.bar(euler_goal_diff['Game Type'], euler_goal_diff['Goal Difference'], color=['blue', 'orange'])
plt.title("Euler's Goal Difference in Home and Away Games")
plt.xlabel('Game Type')
plt.ylabel('Goal Difference')
plt.grid(True)
plt.show()

# Determine if Euler is better in home or away games
if euler_home_goal_diff > euler_away_goal_diff:
    result = "Euler is better in home games."
elif euler_home_goal_diff < euler_away_goal_diff:
    result = "Euler is better in away games."
else:
    result = "Euler performs equally in home and away games."

print(result)

```



Euler is better in home games.

4:

```
[ ] # Calculate goal difference for away players
df_all['away_goal_diff'] = df_all['away_score'] - df_all['home_score']

# Group by away player and calculate the median goal difference
away_goal_diff_median = df_all.groupby('away_player')['away_goal_diff'].median().reset_index()

# Determine the mathematician with the best median away results
best_away_player = away_goal_diff_median.loc[away_goal_diff_median['away_goal_diff'].idxmax()]

print(f"The mathematician with the best median away results is {best_away_player['away_player']} with a median goal difference of {best_away_player['away_goal_diff']}")
```

➡ The mathematician with the best median away results is Von-Neumann with a median goal difference of 5.5.

Part 2

What assumptions do we know for our mathball competition?

We know that there are 6 mathematicians, each year all possible pairs play once.

We have data from the last few years

We also know that in sports scoring is modelled as a Poisson distribution (even in mathball)

We consider home advantage to be a strong effect in sports

The model.

The league is made up by a total of $T = 6$ mathematicians, playing each other once in a season.

We indicate the number of goals scored by the home and the away player in the g -th game of the season (15 games) as y_{g1} and y_{g2} respectively.

The vector of observed goal counts $\mathbf{y} = (y_{g1}, y_{g2})$ is modelled as independent Poisson:

$$y_{gj} | \theta_{gj} \sim \text{Poisson}(\theta_{gj})$$

where the theta parameters represent the scoring intensity in the g -th game for the team playing at home ($j=1$) and away ($j=2$), respectively.

We model these parameters according to a formulation that has been used widely in the statistical literature, assuming a log-linear random effect model:

$$\begin{aligned}\log\theta_{g1} &= \textit{intercept} + \textit{home} + \hat{att}_{h(g)} + \hat{def}_{a(g)} \\ \log\theta_{g2} &= \textit{intercept} + \hat{att}_{a(g)} + \hat{def}_{h(g)}\end{aligned}$$

Alternitevely:

$$\begin{aligned}\theta_{g1} &= e^{\textit{intercept} + \textit{home} + \hat{att}_{h(g)} + \hat{def}_{a(g)}} \\ \theta_{g2} &= e^{\textit{intercept} + \hat{att}_{a(g)} + \hat{def}_{h(g)}}\end{aligned}$$

$\hat{att}_{h(g)}$ is the attack effect of a player.

$\hat{def}_{h(g)}$ is the attack effect of a player.

For mathematician $i = 1, \dots, 6$, both \hat{att}_i and \hat{def}_i are normalized values, fomrally:

$$\begin{aligned}\hat{att}_i &= att_i - \bar{att} \\ \hat{def}_i &= att_i - \bar{def}\end{aligned}$$

where,

$$\begin{aligned}\bar{att} &= \frac{\sum_{i=1}^6 att_i}{6} \\ \bar{def} &= \frac{\sum_{i=1}^6 def_i}{6}\end{aligned}$$

Just for clarification: suppose we index mathematicians as follows:

1: Laplace

2: Gauss

3: Newton

4: Euler

5: Leibniz

6: Von-Neumann

Now, suppose the first game is Laplace hosting Gauss (the clash of the titans).

Thus, $h(1) = 1$ (i.e., Laplace) and $a(1) = 2$ (i.e., Gauss).

The parameter *home* represents the advantage for the team hosting the game and we assume that this effect is constant for all the teams and throughout the season

The scoring intensity is determined jointly by the attack and defense ability of the two teams involved, represented by the parameters att and def , respectively

Conversely, for each $i = 1, \dots, 6$, the player-specific effects are modelled as exchangeable from a common distribution:

$$\text{att}_i \sim \text{Normal}(0, \tau_{\text{att}}) \text{ and } \text{def}_i \sim \text{Normal}(0, \tau_{\text{def}})$$

The log function to away scores and home scores is a standard trick in the sports analytics literature

Tasks

1. Build a Hierarchical model with PyMc where you obtain a posterior of the latent variables: home , att_i , def_i for $i = \{1, 2, 3, 4, 5, 6\}$ and intercept .

Remark: Set the MCMC configuration regarding the number of draws, such that there will 4000 samples in total.

guidance:

For the Hierarchical model, connect the different players (mathematicians) only via the attack and defence std (i.e., τ_{att} and τ_{def}) and not the mean value.

✓ Indexing the data for the pymc model

```
[ ] import numpy as np
```

```
[ ] group = 24
df_all = pickle.load(open('../content/data_mathball/mathball_data_group_' + str(group) + '.pkl', 'rb'))
df_all.head()
```

	home_player	away_player	home_score	away_score	year
0	Laplace	Von-Neumann	27	24	1897
1	Gauss	Euler	26	3	1897
2	Newton	Laplace	17	19	1897
3	Euler	Leibniz	23	15	1897
4	Laplace	Leibniz	31	10	1897

Next steps:

[Generate code with df_all](#)[View recommended plots](#)

▼ Answers to part 2:

```
[ ] ## for your convinience we have here the home and away index with respect to df_all.
home_idx, teams = pd.factorize(df_all["home_player"], sort=True)
away_idx, _ = pd.factorize(df_all["away_player"], sort=True)
```

```
coords = {"team": teams, 'home_idx': home_idx, 'away_idx': away_idx}

with pm.Model(coords=coords) as mathball:

    # Hyperpriors
    tau_att = pm.HalfNormal('tau_att', sigma=1)
    tau_def = pm.HalfNormal('tau_def', sigma=1)

    # Priors
    intercept = pm.Normal('intercept', mu=0, sigma=1)
    home = pm.Normal('home', mu=0, sigma=1)
    att = pm.Normal('att', mu=0, sigma=tau_att, dims="team")
    def_ = pm.Normal('def', mu=0, sigma=tau_def, dims="team")

    ## att_centered function: subtracting the mean att using pm.math.mean (your code here)
    att_centered = att - pm.math.mean(att)
    ## att_centered function: subtracting the mean def using pm.math.mean (your code here)
    def_centered = def_ - pm.math.mean(def_)

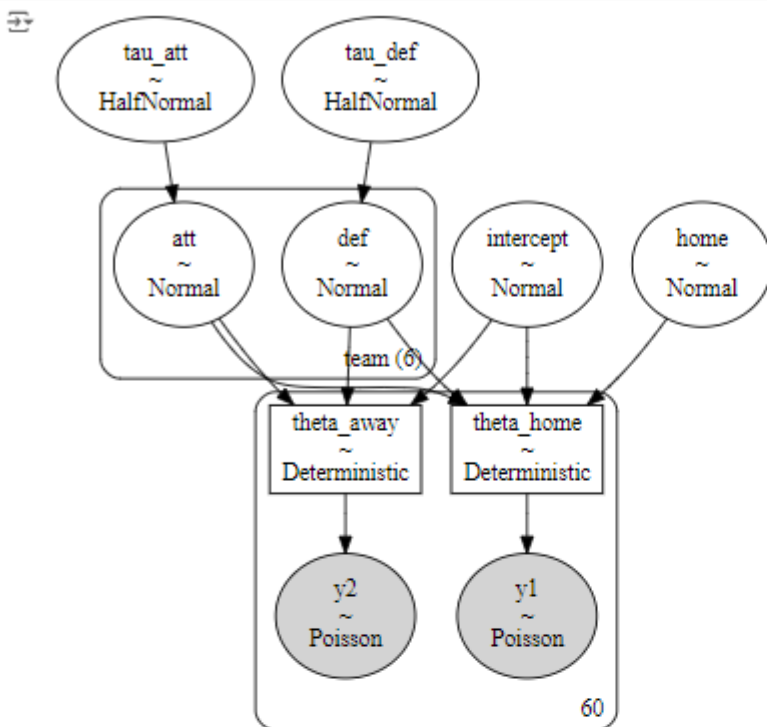
    # theta models
    # log theta_g1 function: intercept + home + att_centered[home_idx] + def_centered[away_idx]
    # exponent function home games (theta_g1) (your code here)
    theta_g1 = pm.Deterministic('theta_home', pm.math.exp(intercept + home + att_centered[home_idx] + def_centered[away_idx]))
    # log theta_g2 function: intercept + att_centered[away_idx] + def_centered[home_idx]
    # exponent function away games (theta_g2) (your code here)
    theta_g2 = pm.Deterministic('theta_away', pm.math.exp(intercept + att_centered[away_idx] + def_centered[home_idx]))

    # Data likelihood
    # Likelihood home score a function theta_g1 - Poisson (your code here)
    y1_likelihood = pm.Poisson('y1', mu=theta_g1, observed=df_all['home_score'].values)
    # Likelihood home score function theta_g2 - Poisson (your code here)
    y2_likelihood = pm.Poisson('y2', mu=theta_g2, observed=df_all['away_score'].values)

    ## pm.sample (your code here)
    trace = pm.sample(2000, tune=1000)
```

```
100.00% [3000/3000 00:24<00:00 Sampling chain 0, 0 divergences]
100.00% [3000/3000 00:25<00:00 Sampling chain 1, 0 divergences]
```

pm.model_to_graphviz(mathball)



Part 3

We aim to understand the different distributions of attacking strength and defensive strength.

These are probabilistic estimates and help us better understand the uncertainty in sports analytics.

Tasks

1. Plot the attack and defence strength HDI for each team and compare them. Which player has the best offence and which has the best defence? who has the worst offence and defence? Does these results is aligned with part 1 analysis?
2. What is the probability that Euler has a better defence than Gauss?
3. Suppose Leibniz is hosting Newton for a game. What is the probability that Leibniz will score more than 20? what is the probability that Newton will score less than 20?
4. Simulate who wins over a total of 4000 simulations, one per sample in the posterior. Hint: use the `sample_posterior_predictive` function.

The resulting simulation should give you the score per each game for all possible combinations for 4 seasons. For each sample out of the 4000 we can measure who will be ranked 1st, 2nd, etc. The ranking is done as follows:

If a player wins he gets 3 points, draw a single point and lost 0 points. Sum up the points and you will get a full ranking. If we aggregate the results, we can the probability for each player for each position.

You are required to draw a bar chart that indicates the probability of all possible ranks for each team.

```
[ ] # from IPython.display import Image, display
    # image_path = 'img_bar_chat.jpg'
    # display(Image(filename=image_path))
```

Your code and answers here

Answers to part 3:

```
[ ] # Extract the posterior samples
att_samples = trace.posterior['att'].values
def_samples = trace.posterior['def'].values

# Calculate the HDI for attack and defense strengths
hdi_att = az.hdi(att_samples, hdi_prob=0.94)
hdi_def = az.hdi(def_samples, hdi_prob=0.94)

# Prepare data for plotting
team_names = teams
att_mean = att_samples.mean(axis=(0, 1))
def_mean = def_samples.mean(axis=(0, 1))

att_hdi_lower = hdi_att[:, 0]
att_hdi_upper = hdi_att[:, 1]
def_hdi_lower = hdi_def[:, 0]
def_hdi_upper = hdi_def[:, 1]

# Create a DataFrame for easier plotting
data_att = pd.DataFrame({
    'Team': team_names,
    'Mean': att_mean,
    'HDI Lower': att_hdi_lower,
    'HDI Upper': att_hdi_upper
})

data_def = pd.DataFrame({
    'Team': team_names,
    'Mean': def_mean,
    'HDI Lower': def_hdi_lower,
    'HDI Upper': def_hdi_upper
})

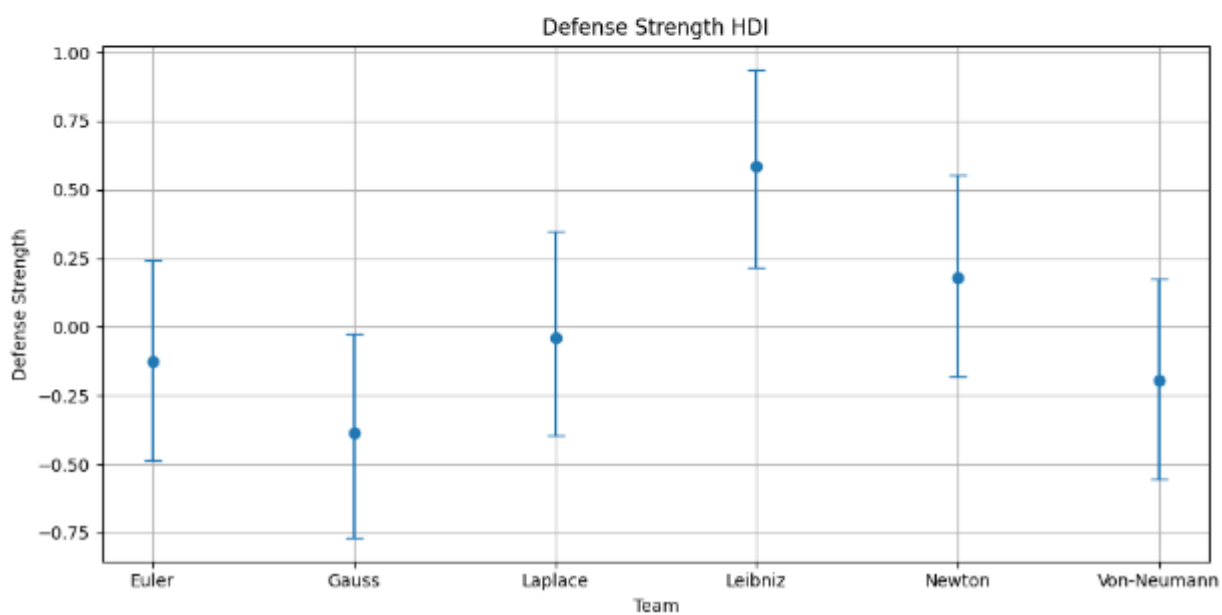
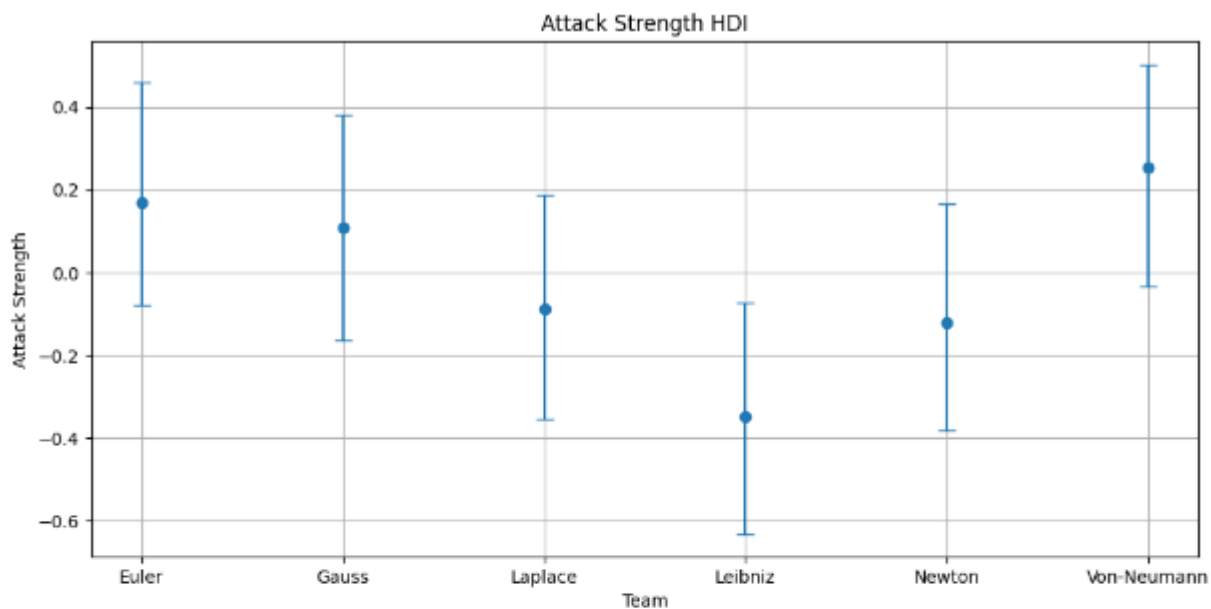
# Plotting
fig, axes = plt.subplots(2, 1, figsize=(10, 10))

# Attack Strength Plot
axes[0].errorbar(data_att['Team'], data_att['Mean'],
                yerr=[data_att['Mean'] - data_att['HDI Lower'], data_att['HDI Upper'] - data_att['Mean']],
                fmt='o', capsize=5)
axes[0].set_title('Attack Strength HDI')
axes[0].set_xlabel('Team')
axes[0].set_ylabel('Attack Strength')
axes[0].grid(True)

# Defense Strength Plot
axes[1].errorbar(data_def['Team'], data_def['Mean'],
                yerr=[data_def['Mean'] - data_def['HDI Lower'], data_def['HDI Upper'] - data_def['Mean']],
                fmt='o', capsize=5)
axes[1].set_title('Defense Strength HDI')
axes[1].set_xlabel('Team')
axes[1].set_ylabel('Defense Strength')
axes[1].grid(True)

plt.tight_layout()
plt.show()

print('Attack Strength df:')
print(data_att)
print('Defense Strength df:')
print(data_def)
```



Attack Strength df:				
	Team	Mean	HDI Lower	HDI Upper
0	Euler	0.170721	-0.078892	0.460800
1	Gauss	0.107890	-0.163289	0.382368
2	Laplace	-0.086513	-0.354747	0.188607
3	Leibniz	-0.349314	-0.631163	-0.074107
4	Newton	-0.122478	-0.380519	0.166616
5	Von-Neumann	0.253119	-0.032445	0.502357

Defense Strength df:				
	Team	Mean	HDI Lower	HDI Upper
0	Euler	-0.123864	-0.486332	0.243932
1	Gauss	-0.387713	-0.768689	-0.025807
2	Laplace	-0.039984	-0.393274	0.348214
3	Leibniz	0.585722	0.214719	0.938342
4	Newton	0.179898	-0.179624	0.552534
5	Von-Neumann	-0.194466	-0.552752	0.175136

```
[ ] # Calculate the mean values of attack and defense parameters
att_mean = att_samples.mean(axis=(0, 1))
def_mean = def_samples.mean(axis=(0, 1))

# Find the indices of the best and worst offense and defense
best_offense_idx = np.argmax(att_mean)
worst_offense_idx = np.argmin(att_mean)
best_defense_idx = np.argmin(def_mean) # Lower is better for defense
worst_defense_idx = np.argmax(def_mean) # Higher is worse for defense

# Get the corresponding player names
best_offense_player = teams[best_offense_idx]
worst_offense_player = teams[worst_offense_idx]
best_defense_player = teams[best_defense_idx]
worst_defense_player = teams[worst_defense_idx]

print(f"The player with the best offense is {best_offense_player}")
print(f"The player with the worst offense is {worst_offense_player}")
print(f"The player with the best defense is {best_defense_player}")
print(f"The player with the worst defense is {worst_defense_player}")
```

```
→ The player with the best offense is Von-Neumann
The player with the worst offense is Leibniz
The player with the best defense is Gauss
The player with the worst defense is Leibniz
```

It seems that the results in this section are not completely aligned with the results we got in part 1. In one part we saw that Leibnitz is the least good player, and in this part we also saw the same result, but in the first part Euler seems to be the best player and here it is divided between Gauss as a defensive player and von Neumann as an attacking player

2:

```
[ ] euler_idx = teams.tolist().index('Euler')
    gauss_idx = teams.tolist().index('Gauss')

# Extract the posterior samples for the defense parameters of Euler and Gauss
euler_def_samples = def_samples[:, :, euler_idx]
gauss_def_samples = def_samples[:, :, gauss_idx]

# Calculate the proportion of samples where Euler's defense is less than Gauss's defense
prob_euler_better_def = (euler_def_samples < gauss_def_samples).mean()

print(f"The probability that Euler has a better defense than Gauss is {prob_euler_better_def:.6f}.")
```

→ The probability that Euler has a better defense than Gauss is 0.000000.

3:

```
● # Get the indices for Leibniz and Newton
leibniz_idx = teams.tolist().index('Leibniz')
newton_idx = teams.tolist().index('Newton')

# Extract the posterior samples for the parameters
intercept_samples = trace.posterior['intercept'].values.flatten()
home_samples = trace.posterior['home'].values.flatten()
att_samples = trace.posterior['att'].values
def_samples = trace.posterior['def'].values

# Compute the expected goals (lambda) for Leibniz (home) and Newton (away)
leibniz_attack = att_samples[:, :, leibniz_idx].flatten()
newton_attack = att_samples[:, :, newton_idx].flatten()
leibniz_defense = def_samples[:, :, leibniz_idx].flatten()
newton_defense = def_samples[:, :, newton_idx].flatten()

# Compute lambda values for Leibniz and Newton
lambda_leibniz = np.exp(intercept_samples + home_samples + leibniz_attack + newton_defense)
lambda_newton = np.exp(intercept_samples + newton_attack + leibniz_defense)

# Ensure lambda values are correctly shaped for the number of simulations
num_simulations = 40000
lambda_leibniz_sim = np.random.choice(lambda_leibniz, size=num_simulations)
lambda_newton_sim = np.random.choice(lambda_newton, size=num_simulations)

# Simulate a large number of game outcomes using the Poisson distribution
leibniz_goals = np.random.poisson(lam=lambda_leibniz_sim)
newton_goals = np.random.poisson(lam=lambda_newton_sim)

# Calculate the probabilities
prob_leibniz_more_than_20 = np.mean(leibniz_goals > 20)
prob_newton_less_than_20 = np.mean(newton_goals < 20)

print(f"The probability that Leibniz will score more than 20 goals is {prob_leibniz_more_than_20:.4f}.")
print(f"The probability that Newton will score less than 20 goals is {prob_newton_less_than_20:.4f}.")
```

→ The probability that Leibniz will score more than 20 goals is 0.3651.
The probability that Newton will score less than 20 goals is 0.1084.

4:


```

import pandas as pd
import matplotlib.pyplot as plt
import arviz as az
import numpy as np

RANDOM_SEED = 58
rng = np.random.default_rng(RANDOM_SEED)

# Sample from the posterior predictive distribution
with mathball:
    ppc = pm.sample_posterior_predictive(trace, extend_inferencedata=True, random_seed=rng)

# Extract and reshape the predicted scores
home_goals = ppc.posterior_predictive["y1"].values.reshape(-1, ppc.posterior_predictive["y1"].shape[-1])
away_goals = ppc.posterior_predictive["y2"].values.reshape(-1, ppc.posterior_predictive["y2"].shape[-1])

# Function to simulate match outcomes and calculate points
def simulate_season(home_goals, away_goals, teams):
    num_games = home_goals.shape[0]
    points = {team: 0 for team in teams}

    for game_idx in range(num_games):
        home_team = teams[home_idx[game_idx]]
        away_team = teams[away_idx[game_idx]]
        home_score = home_goals[game_idx]
        away_score = away_goals[game_idx]

        if home_score > away_score:
            points[home_team] += 3
        elif home_score < away_score:
            points[away_team] += 3
        else:
            points[home_team] += 1
            points[away_team] += 1

    return points

# Simulate the seasons and calculate ranks
num_simulations = home_goals.shape[0]
all_ranks = {team: [] for team in teams}

for i in range(num_simulations):
    points = simulate_season(home_goals[i], away_goals[i], teams)
    sorted_teams = sorted(points.keys(), key=lambda k: points[k], reverse=True)
    for rank, team in enumerate(sorted_teams):
        all_ranks[team].append(rank + 1)

# Convert ranks to a DataFrame
ranks_df = pd.DataFrame({team: pd.Series(ranks) for team, ranks in all_ranks.items()})

# Calculate the probability of each rank for each team
rank_probs = ranks_df.apply(lambda x: x.value_counts(normalize=True)).fillna(0)

# Plot the results
plt.figure(figsize=(15, 8))
width = 0.1 # Width of each bar
colors = plt.cm.get_cmap('tab10', len(teams)).colors # Different colors for each team

for i, team in enumerate(teams):
    rank_probs[team].sort_index().plot(kind='bar', width=width, position=i, color=colors[i], label=team)

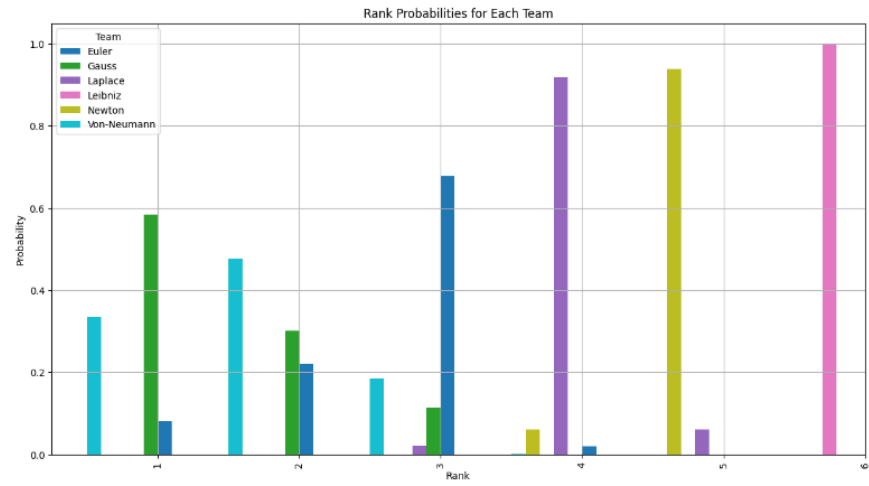
plt.title('Rank Probabilities for Each Team')
plt.xlabel('Rank')
plt.ylabel('Probability')
plt.legend(title='Team')
plt.grid(True)

```


plt.show()

100.00% [4000/4000 00:00<00:00]

MatplotlibDeprecationWarning: The get_cmap function was deprecated in Matplotlib 3.7 and will be removed two minor releases later. Use ``matplotlib.colormaps[name]`` or ``matplotlib.colormaps.get_cmap(obj)`` instead.
colors = plt.cm.get_cmap('tab10', len(teams)).colors # Different colors for each team



queueing_quiz

```
[ ] !pip install simpy
```

Requirement already satisfied: simpy in /usr/local/lib/python3.10/dist-packages (4.1.1)

```
import numpy as np
import pandas as pd
import pickle as pk1
import simpy
import os
import matplotlib.pyplot as plt
import sys
import simpy
from sim_func import Sim_func
import pymc as pm
```

Welcome to the Queue quiz

You are now in charge of the a queueing system in your work place. Your factory works 24/7 and creates flawless semi-conductor chips.

Jobs are arriving with a constant rate, on average one every 12 seconds. However, the service times changes with time. We split the week into a 168 grid, where we have 7 days and 24 hours within a single day. For each cell in the grid, that is a combination of a day and an hour we have different service rate.

Part 1:

We first focus on a single service time of a single cell in the grid. Being more specific, Monday from 08:00 to 09:00.

In event_log_single_ser below you have pandas dataframe with the eventlog of this specific cell.

Your jobs is to understand what is the average service time and its distribution using Bayesian analysis.

```
[ ] event_log_single_ser = pk1.load(open('../content/data_queueing_quiz/event_log_single_ser.pkl', 'rb'))
event_log_single_ser.head()
```

	customer_id	num_cust	event	time_stamp	day	hour
0	0	0	Arrival	30.105252	2	8
1	0	1	Enter_service	30.105252	2	8
2	0	0	Departure	37.798838	2	8
3	1	0	Arrival	38.750399	2	8
4	1	1	Enter_service	38.750399	2	8

Next steps:

[Generate code with event_log_single_ser](#)[View recommended plots](#)

Explanation about the dataframe columns:

1. customer_id: the id of customer, each customer has its own id.
2. num_cust: the number of customers in the system (including service) prior to the event.
3. event: there are three types of events: arrival, entering service and departing the system.
4. Timestamp: The total time elapsed since hour 0 day 1 in seconds.
6. day: the day of the week from 1 to 7 where 1 is Sunday.
7. hour: the hour of the day 0-23, where 0 is midnight.

Tasks:

1. Do Bayesian analysis for the average service rate and its distribution.
2. What did you use as the likelihood function and why?
3. Do posterior predictive analysis to assert that the chosen likelihood distribution is valid.
4. According to the resulted analysis, how certain are you about the average service prediction?

✓ Answers and code

✓ 1. Bayesian analysis for the average service rate and its distribution:

Let's see how the information is defined:

```
[ ] event_log_single_ser.info()
```

```
<class 'pandas.core.frame.DataFrame'>  
RangeIndex: 2189 entries, 0 to 2188  
Data columns (total 6 columns):  
#   Column      Non-Null Count  Dtype  
---  ---  
0   customer_id  2189 non-null   int64  
1   num_cust     2189 non-null   int64  
2   event        2189 non-null   object  
3   time_stamp   2189 non-null   float64  
4   day          2189 non-null   int64  
5   hour         2189 non-null   int64  
dtypes: float64(1), int64(4), object(1)  
memory usage: 182.7+ KB
```

Let's see how many unique customers there are:

```
[ ] event_log_single_ser.groupby('customer_id').sum().shape[0]
```

732

We will create the service rate by the 'Departure' time minus the 'Enter_service':

```
[ ] # Extract service times
enter_service_times = event_log_single_ser[event_log_single_ser['event'] == 'Enter_service'].set_index('customer_id')['time_stamp']
departure_times = event_log_single_ser[event_log_single_ser['event'] == 'Departure'].set_index('customer_id')['time_stamp']

# Calculate service times
service_rate = departure_times - enter_service_times
service_rate = service_rate.dropna()

service_rate
```

customer_id

0	7.693586
1	10.833411
2	2.347220
3	1.187283
4	16.139319
...	
723	2.962984
724	8.557054
725	4.835950
726	19.210130
727	5.067164

Name: time_stamp, Length: 728, dtype: float64

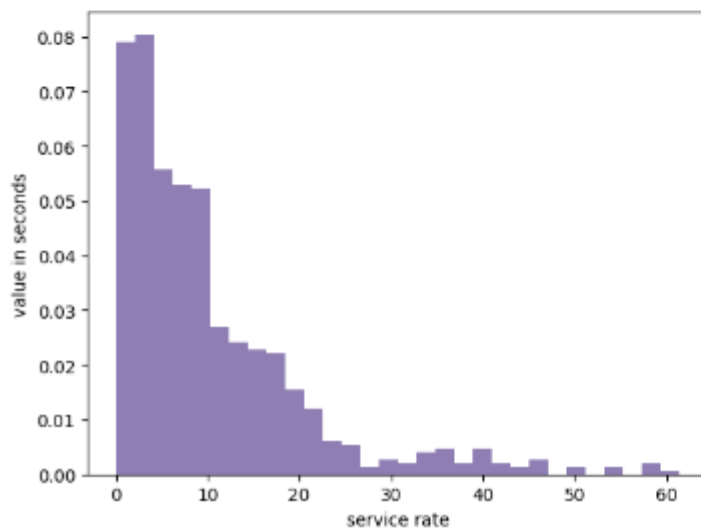
Average service rate and its distribution:

```
[ ] service_rate.mean()
```

10.189650379419726

```
plt.hist(service_rate, histtype='stepfilled', bins=30, alpha=0.85,
         label="service rate in seconds", color="#7A68A6", density=True)
plt.ylabel("value in seconds")
plt.xlabel("service rate")
```

Text(0.5, 0, 'service rate')



Looks like an exponential distribution and since it is about time it also makes sense that this is what we will get

Bayesian analysis:

```
[ ] with pm.Model() as service_rate_model:

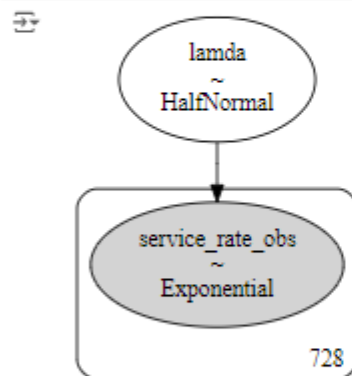
    # Priors for unknown model parameters
    sigma = 1.0/service_rate.mean()
    lamda = pm.HalfNormal('lamda', sigma=sigma)

    # Likelihood (sampling distribution) of observations
    service_rate_obs = pm.Exponential('service_rate_obs', lam=lamda, observed=service_rate)

    # Sample from the posterior
    trace = pm.sample(5000, tune=1000)
```

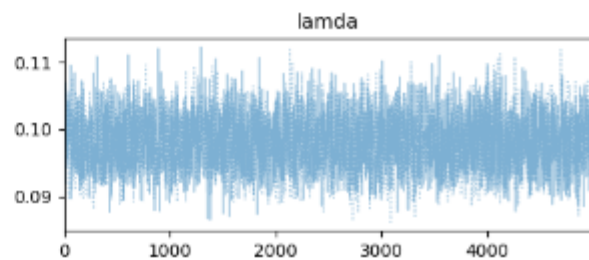
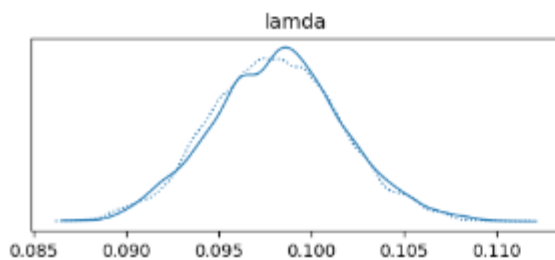
```
100.00% [6000/6000 00:10<00:00 Sampling chain 0, 0 divergences]
100.00% [6000/6000 00:08<00:00 Sampling chain 1, 0 divergences]
```

```
[ ] from pymc import model_to_graphviz
    model_to_graphviz(service_rate_model)
```



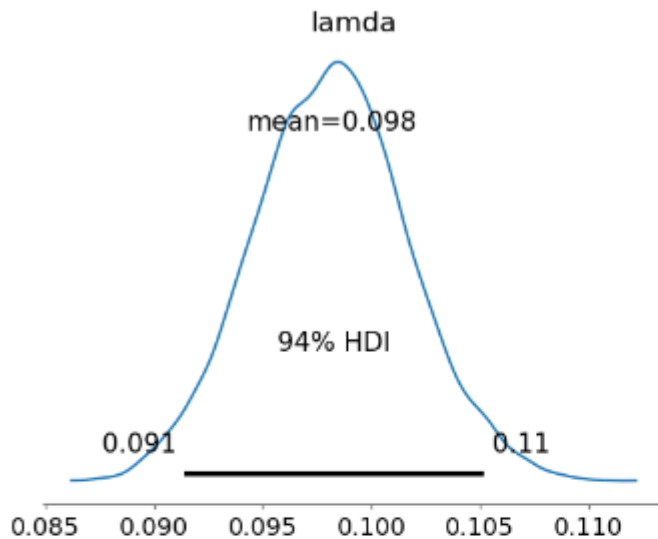
```
import arviz as az
az.plot_trace(trace)
```

```
/usr/local/lib/python3.10/dist-packages/arviz/utils.py:184: NumbaDeprecationWarning: The 'nopython' keyword argument was not sup
numba_fn = numba.jit(**self.kwargs)(self.function)
array([[<Axes: title={'center': 'lamda'}>,
        <Axes: title={'center': 'lamda'}>]], dtype=object)
```



```
az.plot_posterior(trace)
```

```
<Axes: title={'center': 'lamda'}>
```



```
[ ] az.summary(trace)
```

```
/usr/local/lib/python3.10/dist-packages/arviz/utils.py:184: NumbaDeprecationWarning: The 'nopython' keyword argument was not supplied  
numba_fn = numba.jit(**self.kwargs)(self.function)
```

	mean	sd	hdi_3%	hdi_97%	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_hat
lamda	0.098	0.004	0.091	0.105	0.0	0.0	4191.0	6931.0	1.0

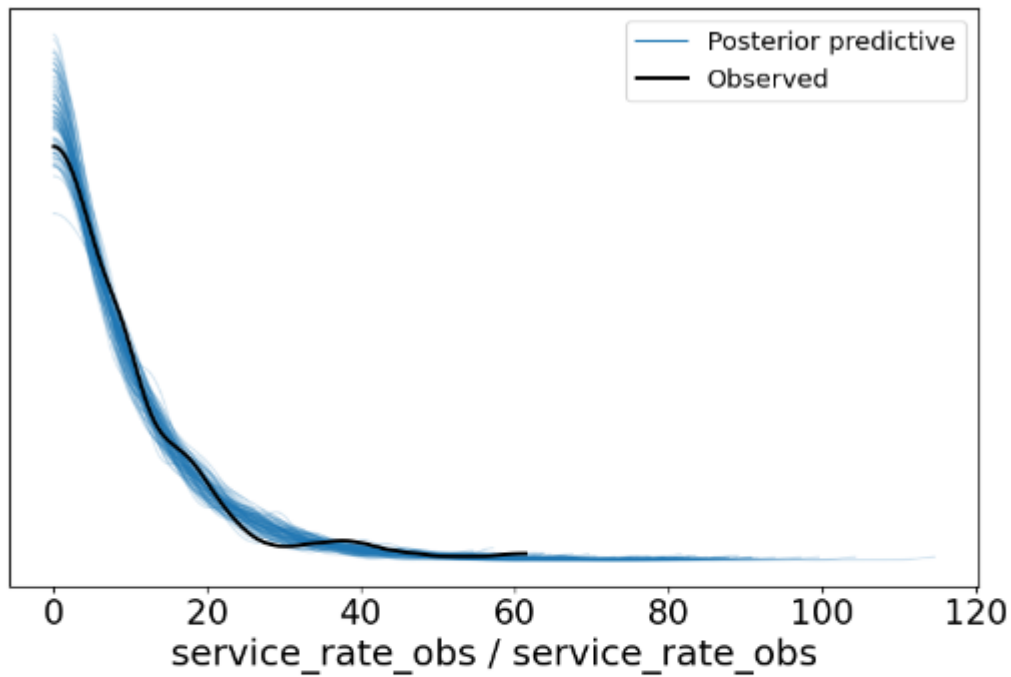
Answer to section 2:

I used an exponential distribution as the likelihood function because first of all it is service time times and also when I made a histogram for the service times it looked like a type of exponential distribution.

3. posterior predictive analysis to assert that the chosen likelihood distribution is valid:

```
[ ] RANDOM_SEED = 58  
rng = np.random.default_rng(RANDOM_SEED)  
  
# Posterior predictive checks  
with service_rate_model:  
    ppc = pm.sample_posterior_predictive(trace, extend_inferencedata=True, random_seed=rng)  
  
ax = az.plot_ppc(ppc, num_pp_samples=100, figsize=(10, 6), mean=False)
```

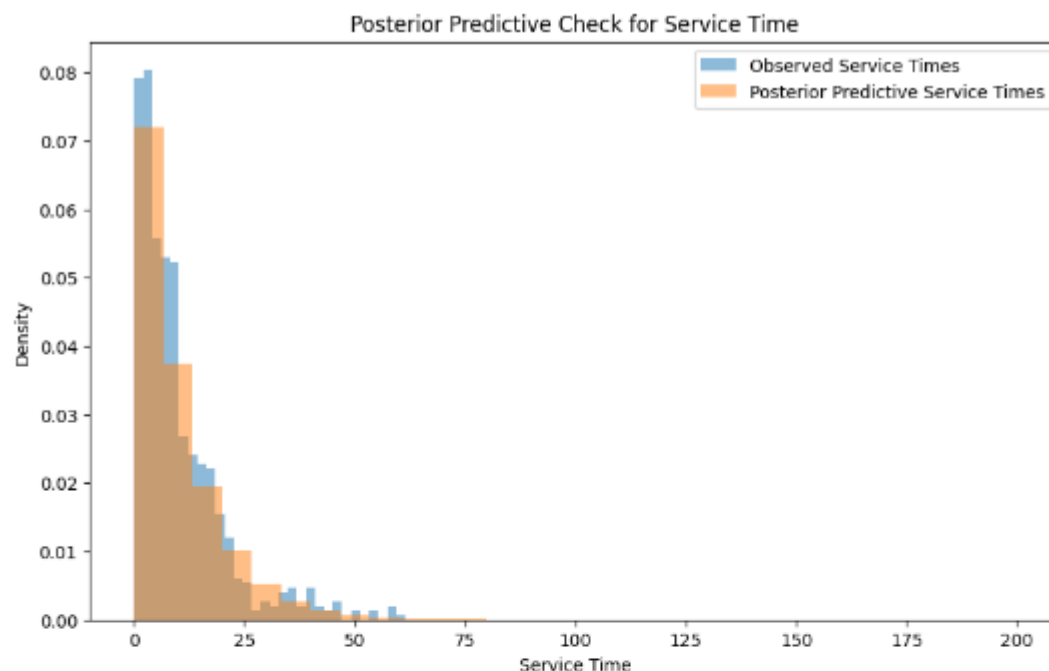
```
100.00% [10000/10000 00:01<00:00]
```



```
# Extract and reshape posterior predictive samples
posterior_predictive_service_times = ppc['posterior_predictive']['service_rate_obs'].values.flatten()

# Compare the distributions
observed_service_times = service_rate.values

plt.figure(figsize=(10, 6))
plt.hist(observed_service_times, bins=30, alpha=0.5, label='Observed Service Times', density=True)
plt.hist(posterior_predictive_service_times, bins=30, alpha=0.5, label='Posterior Predictive Service Times', density=True)
plt.legend(loc='best')
plt.xlabel('Service Time')
plt.ylabel('Density')
plt.title('Posterior Predictive Check for Service Time')
plt.show()
```



4. According to the resulted analysis:

According to the analysis of the results, I wouldn't say that I am completely sure, but I am in the right direction.

The sanity check `plot_trace` comes out good, the two chains converge plus or minus to the same result and there does not seem to be any correlation between the samples. However, in the sanity check `sample_posterior_predictive` there is some small escape of the observations between the number of samples 20-40 which is not very clear to me why it reads and therefore there is no completely normal convergence with posterior predictive. In addition, when comparing the distributions of the posterior predictive and the observations, you do not get full coverage of the posterior predictive on the observations and this can cause us to miss important information. In conclusion, I would not say that there is an unusual incompatibility here that cannot be worked with, but it is possible to further improve the model (not really sure how) in order to have an even better match.

Part 2:

We next turn to the next part where we wish to map the average service time at each time cell.

Important note: please ignore the data from the previous part while doing this task.

The data is under `df_tot` as given below. In the dataset we have missing parts, some cells in the grid are missing. Your job is to fill the grid.


```
[ ] df_tot = pickle.load(open('../content/data_queueing_quiz/df_tot.pkl', 'rb'))
df_tot.head(5)
```

	Unnamed: 0	index	customer_id	num_cust	event	time_stamp	day	hour
0	0	981	327	0	Arrival	3608.653189	1	1
1	1	982	327	1	Enter_service	3608.653189	1	1
2	2	983	327	0	Departure	3609.497913	1	1
3	3	984	328	0	Arrival	3609.790566	1	1
4	4	985	328	1	Enter_service	3609.790566	1	1

Next steps:

[Generate code with df_tot](#)[View recommended plots](#)

Tasks:

1. Extract the mean service times (taking the average time is sufficient, no need for extra bayesian analysis) per each cell you do have data.
2. Use two regression methods for completing the average service time for the rest of the grid.
3. Plot on a 3D plot which contains the data, and the surface of the two regression methods.
4. What is the probability that the service time of chip, between 16:00 to 18:00, on Shabbat, will be between 6 to 7 seconds?
5. Please give full specification of the Gaussian distribution of the service time on Sunday between 14:00 to 15:00 and Sunday between 16:00 to 17:00.
6. What is the correlation between the service times at Sunday between 14:00 to 15:00 and Sunday between 16:00 to 17:00?

What is the correlation between the service times at Sunday between 14:00 to 15:00 and Sunday between 20:00 to 21:00?

Which correlation value is larger? does this result make sense?

✓ You code here

✓ Answers to part 2:

1. mean service times per each cell we do have data:

```
[ ] # Calculate service times
enter_service = df_tot[df_tot['event'] == 'Enter_service'].set_index('customer_id')['time_stamp']
departure = df_tot[df_tot['event'] == 'Departure'].set_index('customer_id')['time_stamp']



service_times = departure - enter_service

# Group by day and hour, calculate mean service time
mean_service_times = service_times.groupby([df_tot['day'], df_tot['hour']]).mean()

# Create a DataFrame with all possible day-hour combinations in the grid
days = range(1, 8) # days range from 1 to 7
hours = range(0, 24) # hours range from 0 to 23
day_hour_combinations = [(day, hour) for day in days for hour in hours]

# Convert to DataFrame
mean_service_times_df = pd.DataFrame(index=pd.MultiIndex.from_tuples(day_hour_combinations, names=['day', 'hour']))
mean_service_times_df['mean_service_time'] = mean_service_times

# Display the resulting DataFrame
mean_service_times_df.head()
```


 **mean_service_time** 

day	hour	
1	0	NaN
	1	1.293141
	2	NaN
	3	1.893939
	4	2.082484

Next steps:

[Generate code with mean_service_times_df](#)

[View recommended plots](#)

 `mean_service_times_df['mean_service_time'].mean()`

 3.654658933222642

```

import plotly.graph_objects as go
import numpy as np
import pandas as pd

# Reset index to access day and hour as columns
mean_service_times_df = mean_service_times_df.reset_index()

# Extract data for plotting
days = mean_service_times_df['day']
hours = mean_service_times_df['hour']
mean_service_times = mean_service_times_df['mean_service_time']

# Create a 3D scatter plot
fig = go.Figure(data=[go.Scatter3d(
    x=days,
    y=hours,
    z=mean_service_times,
    mode='markers',
    marker=dict(
        size=5,
        color=mean_service_times, # Set color to the mean service times
        colorscale='Viridis', # Choose a colorscale
        opacity=0.8
    )
)])

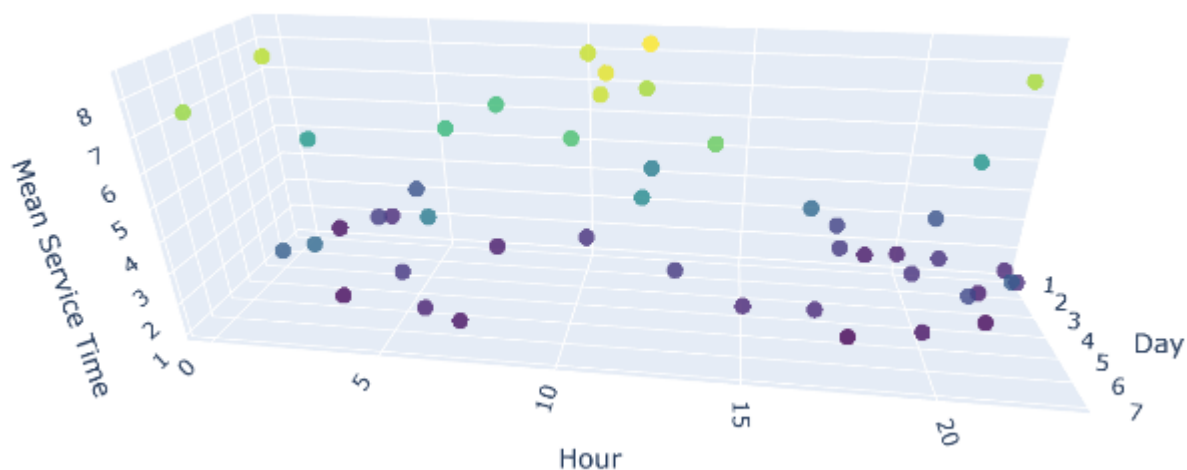
# Set labels for axes
fig.update_layout(
    scene=dict(
        xaxis_title='Day',
        yaxis_title='Hour',
        zaxis_title='Mean Service Time'
    ),
    title='Interactive 3D Scatter Plot of Mean Service Times'
)

# Show the plot
fig.show()

```

(1)

Interactive 3D Scatter Plot of Mean Service Times



2. Using two regression methods for completing the average service time for the rest of the grid:

Division into training and test (when there is no Y test).

Training - the full cells.

Test - the empty cells.

```
import pandas as pd

# Calculate service times
df_tot = df_tot.copy()

# Create a dictionary to store service times for each customer
service_times = {}

# Iterate over each row in the DataFrame
for _, row in df_tot.iterrows():
    customer_id = row['customer_id']
    event = row['event']
    time_stamp = row['time_stamp']

    if event == 'Enter_service':
        if customer_id not in service_times:
            service_times[customer_id] = {}
        service_times[customer_id]['enter_service'] = time_stamp
    elif event == 'Departure':
        if customer_id in service_times:
            service_times[customer_id]['departure'] = time_stamp

# Calculate service times and add them to the DataFrame
service_times_list = []
for customer_id, times in service_times.items():
    if 'enter_service' in times and 'departure' in times:
        service_time = times['departure'] - times['enter_service']
        service_times_list.append({
            'customer_id': customer_id,
            'service_time': service_time,
            'day': df_tot[df_tot['customer_id'] == customer_id]['day'].values[0],
            'hour': df_tot[df_tot['customer_id'] == customer_id]['hour'].values[0]
        })

service_times_df = pd.DataFrame(service_times_list)

# Aggregate service times by day and hour
mean_service_times_df = service_times_df.groupby(['day', 'hour'])['service_time'].mean().reset_index()

# Identify empty and filled cells
days = range(1, 8) # days range from 1 to 7
hours = range(0, 24) # hours range from 0 to 23
day_hour_combinations = pd.DataFrame([(day, hour) for day in days for hour in hours], columns=['day', 'hour'])

# Merge to find empty cells
merged_df = pd.merge(day_hour_combinations, mean_service_times_df, on=['day', 'hour'], how='left')

# Identify empty and filled cells
empty_cells = merged_df[merged_df['service_time'].isna()]
filled_cells = merged_df[~merged_df['service_time'].isna()]
```

```
# Prepare training and test sets
X_train = filled_cells[['day', 'hour']]
y_train = filled_cells['service_time']
X_test = empty_cells[['day', 'hour']]

print("Training Set (Filled Cells):")
print(X_train.head())
print(y_train.head())

print("Test Set (Empty Cells):")
print(X_test.head())
```

```
Training Set (Filled Cells):
   day  hour
1     1     1
3     1     3
4     1     4
7     1     7
10    1    10
1  1.293141
3  1.729359
4  2.063790
7  2.982484
10 5.594943
Name: service_time, dtype: float64
Test Set (Empty Cells):
   day  hour
0     1     0
2     1     2
5     1     5
6     1     6
8     1     8
```

linear regression:

```
[ ] # Standardize the data
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

with pm.Model() as linear_model:
    # Priors for unknown model parameters
    beta = pm.Normal('beta', mu=0, sigma=50, shape=2)
    sigma = pm.HalfNormal('sigma', sigma=10)

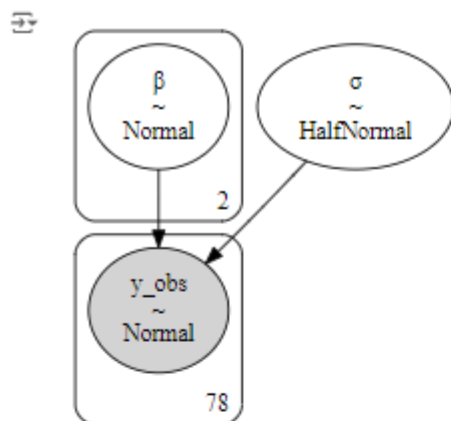
    # Linear function
    mu = beta[0] + beta[1] * X_train_scaled[:, 1]

    # Likelihood (sampling distribution) of observations
    y_obs = pm.Normal('y_obs', mu=mu, sigma=sigma, observed=y_train)

    # Sample from the posterior
    trace_linear = pm.sample(2000, tune=1000, chains=2, cores=1)
```

```
100.00% [3000/3000 00:02<00:00 Sampling chain 0, 0 divergences]
100.00% [3000/3000 00:03<00:00 Sampling chain 1, 0 divergences]
```

```
[ ] pm.model_to_graphviz(linear_model)
```



```
[ ] # Posterior predictive check
with linear_model:
    linear_pred = pm.sample_posterior_predictive(trace_linear, var_names=['beta'])
```

100.00% [4000/4000 00:00<00:00]

```
[ ] # Extract the predictions
beta_samples = linear_pred.posterior_predictive['beta'].values
pred_mean_linear = np.mean(beta_samples[:, 0].reshape(-1, 1) + beta_samples[:, 1].reshape(-1, 1) * X_test_scaled[:, 1], axis=0)
```

```
# Fill the empty cells with the Linear Regression predictions
empty_cells.loc[:, 'service_time'] = pred_mean_linear

# Combine filled and empty cells
filled_grid_linear = pd.concat([filled_cells, empty_cells], ignore_index=True)
filled_grid_linear.head()
```

	day	hour	service_time
0	1	1	1.293141
1	1	3	1.729359
2	1	4	2.063790
3	1	7	2.982484
4	1	10	5.594943

Next steps: [Generate code with filled_grid_linear](#)

[View recommended plots](#)

Gaussian process (GP)

According to the three-dimensional graph we made in part 2, section 1, it seems that the information can have certain fluctuations, so we chose the Matern32 kernel.

```
[ ] # Standardize the data
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_train_scaled = np.asarray(X_train_scaled)
y_train = np.asarray(y_train)
X_test_scaled = scaler.transform(X_test)
X_test_scaled = np.asarray(X_test_scaled)
```

```
with pm.Model() as gp_model:

    ## Defining the priors
    l = pm.HalfCauchy("l", beta=3, shape=(2,))
    sf2 = pm.HalfCauchy("sf2", beta=3)

    ## 6. Defining the kernel
    kenral_option = 2 # The information can have certain fluctuations
    # option 1: expquad
    if kenral_option == 1:
        K = pm.gp.cov.ExpQuad(2, 1) * sf2**2
    elif kenral_option == 2:
        # option 2: Matern32
        K = pm.gp.cov.Matern32(2, 1) * sf2**2
    elif kenral_option == 3:
        # option 3: Matern 52
        K = pm.gp.cov.Matern52(2, 1) * sf2**2

    # 5. Mean functions
    mean_func = pm.gp.mean.Linear(coeffs=[1, 1])

    ## 4. Defining the GP process
    gp_priors = pm.gp.Marginal(mean_func=mean_func, cov_func=K)

    ## 3. Prior of the noise
    sigma = pm.HalfCauchy('sigma', beta=5)

    ## 2. Defining the likelihood
    y_obs = gp_priors.marginal_likelihood('y_obs', X=X_train_scaled, y=y_train, noise=sigma)

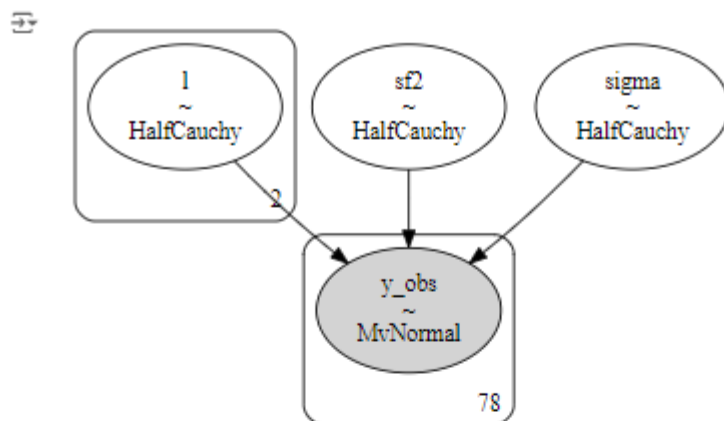
    ## 1. sampling ncmc
    trace_gp = pm.sample(1000, tune=1000, chains=2, cores=1)
```

 /usr/local/lib/python3.10/dist-packages/pymc/gp/gp.py:56: FutureWarning:

The 'noise' parameter has been changed to 'sigma' in order to standardize the GP API and will be deprecated in future releases.

```
100.00% [2000/2000 01:30<00:00 Sampling chain 0, 0 divergences]
100.00% [2000/2000 01:18<00:00 Sampling chain 1, 0 divergences]
```

```
[ ] pm.model_to_graphviz(gp_model)
```



```
[ ] # Posterior predictive check
with gp_model:
    f_pred = gp_priors.conditional('f_pred', X_test_scaled)
    pred_samples = pm.sample_posterior_predictive(trace_gp, var_names=['f_pred'])
```

100.00% [2000/2000 00:32<00:00]

```
[ ] # Extract the predictions
pred_mean_gp = np.mean(pred_samples.posterior_predictive['f_pred'], axis=(0, 1))
```

```
# Fill the empty cells with the GP predictions
empty_cells.loc[:, 'service_time'] = pred_mean_gp

# Combine filled and empty cells
filled_grid_gp = pd.concat([filled_cells, empty_cells], ignore_index=True)

filled_grid_gp.tail()
```

	day	hour	service_time
163	7	14	7.983439
164	7	15	7.489534
165	7	16	6.428806
166	7	17	5.175490
167	7	20	2.431501

3. A 3D plot which contains the data, and the surface of the two regression methods:


```

import numpy as np
import plotly.graph_objects as go

# Define the correct column name
service_time_col = 'service_time'

# Extract the service time column from the DataFrames
filled_grid_linear_np = filled_grid_linear[service_time_col].to_numpy()
filled_grid_gp_np = filled_grid_gp[service_time_col].to_numpy()

print("filled_grid_linear_np shape before reshape:", filled_grid_linear_np.shape)
print("filled_grid_gp_np shape before reshape:", filled_grid_gp_np.shape)

# Create a mesh grid for the surface plots
x = np.arange(mean_service_times_df['hour'].min(), mean_service_times_df['hour'].max() + 1, 1)
y = np.arange(mean_service_times_df['day'].min(), mean_service_times_df['day'].max() + 1, 1)
X, Y = np.meshgrid(x, y)

# Flatten the mesh grid to match the data for plotting
x_flat = X.flatten()
y_flat = Y.flatten()

# Ensure the filled grids have the correct size before reshaping
expected_size = X.size
print("Expected size:", expected_size)

if filled_grid_linear_np.size == expected_size:
    filled_grid_linear_np = filled_grid_linear_np.reshape(Y.shape)
else:
    print("filled_grid_linear size does not match the expected size.")

if filled_grid_gp_np.size == expected_size:
    filled_grid_gp_np = filled_grid_gp_np.reshape(Y.shape)
else:
    print("filled_grid_gp size does not match the expected size.")

# Create a 3D scatter plot for the actual data points
scatter_data = go.Scatter3d(
    x=mean_service_times_df['hour'],
    y=mean_service_times_df['day'],
    z=mean_service_times_df['service_time'],
    mode='markers',
    marker=dict(
        size=5,
        color=mean_service_times_df['service_time'],
        colorscale='Viridis',
        opacity=0.8
    ),
    name='Data Points'
)

# Create 3D surface plot for Linear Regression
if filled_grid_linear_np.size == expected_size:
    surface_linear = go.Surface(
        x=X,
        y=Y,
        z=filled_grid_linear_np,
        colorscale='Viridis',
        opacity=0.7,
        name='Linear Regression Surface'
    )
else:
    surface_linear = None

```

```

# Create 3D surface plot for GP Regression
if filled_grid_gp_np.size == expected_size:
    surface_gp = go.Surface(
        x=X,
        y=Y,
        z=filled_grid_gp_np,
        colorscale='Plasma',
        opacity=0.7,
        name='GP Regression Surface'
    )
else:
    surface_gp = None

# Combine the plots
fig = go.Figure(data=[scatter_data])

if surface_linear is not None:
    fig.add_trace(surface_linear)

if surface_gp is not None:
    fig.add_trace(surface_gp)

# Set labels for axes
fig.update_layout(
    scene=dict(
        xaxis_title='Hour',
        yaxis_title='Day',
        zaxis_title='Service Time',
        aspectmode='manual',
        aspectratio=dict(x=1, y=1, z=1)
    ),
    title='Service Time Prediction',
    width=700,
    height=700
)

# Show the plot
fig.show()

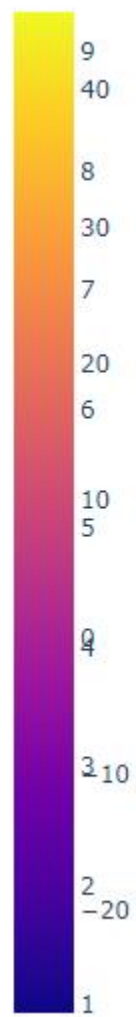
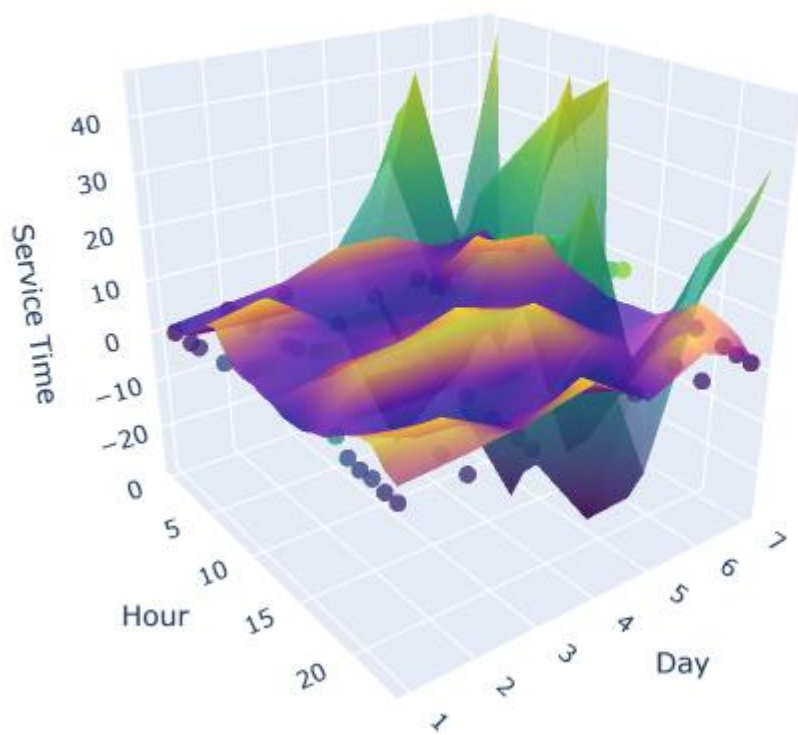
```

```

→ filled_grid_linear_np shape before reshape: (168,)
filled_grid_gp_np shape before reshape: (168,)
Expected size: 168

```

Service Time Prediction



Unsurprisingly the Gaussian process looks better.

4. The probability that the service time of chip, between 16:00 to 18:00, on Shabbat, will be between 6 to 7 seconds:

```
[ ] # Filter for the time between 16:00 and 18:00 on Shabbat
# Adjust 'day' value if necessary for Shabbat (day=7 represents Shabbat)
filtered_df = filled_grid_gp[(filled_grid_gp['hour'] >= 16) & (filled_grid_gp['hour'] <= 18) & (filled_grid_gp['day'] == 7)]

# Calculate the probability of service time being between 6 and 7 seconds
lower_bound = 6
upper_bound = 7

# Find the rows where service_time is between 6 and 7 seconds
within_range = (filtered_df['service_time'] >= lower_bound) & (filtered_df['service_time'] <= upper_bound)

# Compute the probability
probability_within_range = np.mean(within_range)

# Print the result
print(f'Probability that the service time is between {lower_bound} and {upper_bound} seconds between 16:00 and 18:00 on Shabbat: {probability_within_range}')
```

➡ Probability that the service time is between 6 and 7 seconds between 16:00 and 18:00 on Shabbat: 0.3333333333333333

5. Full specification of the Gaussian distribution of the service time on Sunday between 14:00 to 15:00 and Sunday between 16:00 to 17:00:

```
[ ] import numpy as np
import pandas as pd
from sklearn.preprocessing import StandardScaler

# Extract relevant rows from filled_grid_gp for Sunday (day=1) and hours 14:00 to 15:00 and 16:00 to 17:00
sunday_data = filled_grid_gp[(filled_grid_gp['day'] == 1) & (filled_grid_gp['hour'].isin([14, 16]))]
sunday_data = sunday_data.sort_values(by=['hour'])

# Extract the service times, hours, and days
service_times = sunday_data['service_time'].values
hours = sunday_data['hour'].values
days = sunday_data['day'].values

# Combine hours and days into a single array for GP input
X = np.column_stack([days, hours])

# Standardize the features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
```

```

# Fit the Gaussian Process model
with pm.Model() as gp_model:

    # Define the priors
    l = pm.HalfCauchy("l", beta=3, shape=(2,))
    sf2 = pm.HalfCauchy("sf2", beta=3)

    # Define the kernel
    kernel_option = 2
    if kernel_option == 1:
        K = pm.gp.cov.ExpQuad(2, 1) * sf2**2
    elif kernel_option == 2:
        K = pm.gp.cov.Matern32(2, 1) * sf2**2
    elif kernel_option == 3:
        K = pm.gp.cov.Matern52(2, 1) * sf2**2

    # Define the mean function
    mean_func = pm.gp.mean.Linear(coeffs=[1, 1])

    # Define the GP process
    gp_priors = pm.gp.Marginal(mean_func=mean_func, cov_func=K)

    # Define the noise prior
    sigma = pm.HalfCauchy('sigma', beta=5)

    # Define the likelihood
    y_obs = gp_priors.marginal_likelihood('y_obs', X=X_scaled, y=service_times, noise=sigma)

    # Sample from the posterior
    trace_gp = pm.sample(1000, tune=1000, chains=2, cores=1, return_inferencedata=True)

```

 /usr/local/lib/python3.10/dist-packages/pymc/gp/gp.py:56: FutureWarning:

The 'noise' parameter has been changed to 'sigma' in order to standardize the GP API and will be deprecated in future releases.

```

100.00% [2000/2000 00:17<00:00 Sampling chain 0, 5 divergences]
100.00% [2000/2000 00:21<00:00 Sampling chain 1, 1 divergences]

```

```

[ ] # Define the time points we are interested in (hours 14, 16) for Sunday (day=1)
test_hours = np.array([14, 16])
test_days = np.array([1, 1]) # Sundays

# Combine test hours and days into a single array
X_test = np.column_stack([test_days, test_hours])

# Standardize the test features using the same scaler
X_test_scaled = scaler.transform(X_test)

# Predict the mean and covariance using the GP model
with gp_model:
    mu, cov = gp_priors.predict(X_test_scaled, point=pm.find_MAP())

# Print the results
print("Mu vector:", mu)
print("Covariance Matrix:", cov)

```

 100.00% [28/28 00:00<00:00 logp = -14.996, ||grad|| = 0.0015588]

```

Mu vector: [-0.99999993  1.00000005]
Covariance Matrix: [[1.30927303e-06 0.00000000e+00]
 [0.00000000e+00 1.30927303e-06]]

```

6. Which correlation value is larger?

The correlation between the service times at Sunday between 14:00 to 15:00 and Sunday between 16:00 to 17:00 or the correlation between the service times at Sunday between 14:00 to 15:00 and Sunday between 20:00 to 21:00.

Does this result make sense?

```
import numpy as np
import pandas as pd
from sklearn.preprocessing import StandardScaler

# Extract relevant rows from filled_grid_gp for Sunday (day=1) and hours 14:00 to 15:00, 16:00 to 17:00, and 20:00 to 21:00
sunday_data = filled_grid_gp[(filled_grid_gp['day'] == 1) & (filled_grid_gp['hour'].isin([14, 16, 20]))]
sunday_data = sunday_data.sort_values(by=['hour'])

# Extract the service times, hours, and days
service_times = sunday_data['service_time'].values
hours = sunday_data['hour'].values
days = sunday_data['day'].values

# Combine hours and days into a single array for GP input
X = np.column_stack([days, hours])

# Standardize the features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Fit the Gaussian Process model
with pm.Model() as gp_model:

    # Define the priors
    l = pm.HalfCauchy("l", beta=3, shape=(2,))
    sf2 = pm.HalfCauchy("sf2", beta=3)

    # Define the kernel
    kernel_option = 2
    if kernel_option == 1:
        K = pm.gp.cov.ExpQuad(2, 1) * sf2**2
    elif kernel_option == 2:
        K = pm.gp.cov.Matern32(2, 1) * sf2**2
    elif kernel_option == 3:
        K = pm.gp.cov.Matern52(2, 1) * sf2**2

    # Define the mean function
    mean_func = pm.gp.mean.Linear(coeffs=[1, 1])

    # Define the GP process
    gp_priors = pm.gp.Marginal(mean_func=mean_func, cov_func=K)

    # Define the noise prior
    sigma = pm.HalfCauchy('sigma', beta=5)


    # Define the likelihood
    y_obs = gp_priors.marginal_likelihood('y_obs', X=X_scaled, y=service_times, noise=sigma)

    # Sample from the posterior
    trace_gp = pm.sample(1000, tune=1000, chains=2, cores=1, return_inferencedata=False)

# Define the time points we are interested in
test_hours = np.array([14, 16, 20])
test_days = np.array([1, 1, 1]) # Sundays

# Combine test hours and days into a single array
X_test = np.column_stack([test_days, test_hours])
```

```
# Standardize the test features using the same scaler
X_test_scaled = scaler.transform(X_test)
```

 /usr/local/lib/python3.10/dist-packages/pymc/gp/gp.py:56: FutureWarning:

The 'noise' parameter has been changed to 'sigma' in order to standardize the GP API and will be deprecated in future releases.


```
100.00% [2000/2000 00:21<00:00 Sampling chain 0, 2 divergences]
100.00% [2000/2000 00:22<00:00 Sampling chain 1, 2 divergences]
```

```
[ ] # Predict the mean and covariance using the GP model
    with gp_model:
        mu, cov = gp_priors.predict(X_test_scaled, point=pm.find_MAP())

    # Calculate correlations
    corr_14_16 = cov[0, 1] / (np.sqrt(cov[0, 0] * cov[1, 1]))
    corr_14_20 = cov[0, 2] / (np.sqrt(cov[0, 0] * cov[2, 2]))

    # Print the results
    print("Correlation between service times at Sunday 14:00 to 15:00 and Sunday 16:00 to 17:00:", corr_14_16)
    print("Correlation between service times at Sunday 14:00 to 15:00 and Sunday 20:00 to 21:00:", corr_14_20)

    if corr_14_16 > corr_14_20:
        print("The correlation between service times at Sunday 14:00 to 15:00 and Sunday 16:00 to 17:00 is larger.")
    else:
        print("The correlation between service times at Sunday 14:00 to 15:00 and Sunday 20:00 to 21:00 is larger.")
```

 100.00% [30/30 00:00<00:00 logp = -17.346, ||grad|| = 0.02827]

```
Correlation between service times at Sunday 14:00 to 15:00 and Sunday 16:00 to 17:00: 5.47624345436991e-07
Correlation between service times at Sunday 14:00 to 15:00 and Sunday 20:00 to 21:00: -1.0372576474037422e-07
The correlation between service times at Sunday 14:00 to 15:00 and Sunday 16:00 to 17:00 is larger.
```

It makes sense that the correlation between two consecutive hours is higher than two hours between which there is an interval, when it is a plant that provides a service.

It can be thought of like a store where people arrive in "waves" and therefore when there is no time interval the correlation is greater.


▼ Task 3:

For cross validation of the previous task we wish to see how accurate our predictions.

We use your prediction to estimate the waiting times at each points via simulation. The better the estimation made in you GP regressor the more accurate the simulations results.

In this part we compare the average waiting time at each cell with the true average waiting times. The true values are loaded below into 'true_results'.

```
[ ] true_results = pickle.load(open('../content/data_queueing_quiz/true_results.pkl', 'rb'))
    true_results.shape
```

 (167,)

✓ In true_results you have a numpy array with 167 values. The first value is the average number of customers in the system at the end of the hour 0 day 1, the second is hour 1 day 1, and so on. This is an average of 200 simulation runs. Note that the grid contains 168 values but here we have only 167. This means, we do not test for 23:00-0:00 at the day 7 (from technical reasons).

In order to compute your waiting time predictions insert your predictions into Sim_func below. Then, the line of code below extracts the average waiting time predictions into the variable 'preds_results'.

At this point you have two vectors of size (167), one represents the true average waiting times and the other represent your predictions.

The first value represents the average waiting time at Sunday between 0:00-01:00. The 50th cell represent the average waiting time at Tuesday between 01:00 to 02:00.

Tasks

1. Compute the average waiting times using the function Sim_func. Compare your results between your predictions and the ground truth using MSE.
2. Plot the average waiting as a function of time of the week. That is, y-axis is the average waiting time and the X-axis is the hour of the week. Following the example from above Tuesday between 01:00 to 02:00 is the 50th hour.

```
[ ] ## Insert your predictions her

# df_ = Sim_func(YOUR_PREDICTION)
# ## Converting the simulation results into a numpy array from a PD dataframe.
# preds_results = np.array(df_.iloc[:,2:]).mean(axis = 1)
```

✓ Answers to part 3:

1. Computing the average waiting times using the function Sim_func, and Comparing my results between your predictions and the ground truth using MSE:

We will repeat the distribution of the information and the model.


```
[ ] import pandas as pd

# Calculate service times
df_tot = df_tot.copy()

# Create a dictionary to store service times for each customer
service_times = {}

# Iterate over each row in the DataFrame
for _, row in df_tot.iterrows():
    customer_id = row['customer_id']
    event = row['event']
    time_stamp = row['time_stamp']

    if event == 'Enter_service':
        if customer_id not in service_times:
            service_times[customer_id] = {}
        service_times[customer_id]['enter_service'] = time_stamp
    elif event == 'Departure':
        if customer_id in service_times:
            service_times[customer_id]['departure'] = time_stamp

# Calculate service times and add them to the DataFrame
service_times_list = []
for customer_id, times in service_times.items():
    if 'enter_service' in times and 'departure' in times:
        service_time = times['departure'] - times['enter_service']
        service_times_list.append({
            'customer_id': customer_id,
            'service_time': service_time,
            'day': df_tot[df_tot['customer_id'] == customer_id]['day'].values[0],
            'hour': df_tot[df_tot['customer_id'] == customer_id]['hour'].values[0]
        })

service_times_df = pd.DataFrame(service_times_list)

# Aggregate service times by day and hour
mean_service_times_df = service_times_df.groupby(['day', 'hour'])['service_time'].mean().reset_index()

# Identify empty and filled cells
days = range(1, 8) # days range from 1 to 7
hours = range(0, 24) # hours range from 0 to 23
day_hour_combinations = pd.DataFrame([(day, hour) for day in days for hour in hours], columns=['day', 'hour'])

# Merge to find empty cells
merged_df = pd.merge(day_hour_combinations, mean_service_times_df, on=['day', 'hour'], how='left')

# Identify empty and filled cells
empty_cells = merged_df[merged_df['service_time'].isna()]
filled_cells = merged_df[~merged_df['service_time'].isna()]

# Prepare training and test sets
X_train = filled_cells[['day', 'hour']]
y_train = filled_cells['service_time']
X_test = empty_cells[['day', 'hour']]

print("Training Set (Filled Cells):")
print(X_train.head())
print(y_train.head())

print("Test Set (Empty Cells):")
print(X_test.head())
```

```

# Standardize the data
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_train_scaled = np.asarray(X_train_scaled)
y_train = np.asarray(y_train)
X_test_scaled = scaler.transform(X_test)
X_test_scaled = np.asarray(X_test_scaled)
with pm.Model() as gp_model:

    ## Defining the priors
    l = pm.HalfCauchy("l", beta=3, shape=(2,))
    sf2 = pm.HalfCauchy("sf2", beta=3)

    ## 6. Defining the kernel
    kenral_option = 2 # The information can have certain fluctuations
    # option 1: expquad
    if kenral_option == 1:
        K = pm.gp.cov.ExpQuad(2, 1) * sf2**2
    elif kenral_option == 2:
        # option 2: Matern32
        K = pm.gp.cov.Matern32(2, 1) * sf2**2
    elif kenral_option == 3:
        # option 3: Matern 52
        K = pm.gp.cov.Matern52(2, 1) * sf2**2

    # 5. Mean functions
    mean_func = pm.gp.mean.Linear(coeffs=[1, 1])

    ## 4. Defining the GP process
    gp_priors = pm.gp.Marginal(mean_func=mean_func, cov_func=K)

    ## 3. Prior of the noise
    sigma = pm.HalfCauchy('sigma', beta=5)

    ## 2. Defining the likelihood
    y_obs = gp_priors.marginal_likelihood('y_obs', X=X_train_scaled, y=y_train, noise=sigma)

    ## 1. sampling mcmc
    trace_gp = pm.sample(1000, tune=1000, chains=2, cores=1)

# Posterior predictive check
with gp_model:
    f_pred = gp_priors.conditional('f_pred', X_test_scaled)
    pred_samples = pm.sample_posterior_predictive(trace_gp, var_names=['f_pred'])

# Extract the predictions
pred_mean_gp = np.mean(pred_samples.posterior_predictive['f_pred'], axis=(0, 1))

# Fill the empty cells with the GP predictions
empty_cells.loc[:, 'service_time'] = pred_mean_gp

# Combine filled and empty cells
filled_grid_gp = pd.concat([filled_cells, empty_cells], ignore_index=True)

filled_grid_gp.tail()

```

Training Set (Filled Cells):

	day	hour
1	1	1
3	1	3
4	1	4
7	1	7
10	1	10
1	1.293141	
3	1.729359	
4	2.063790	
7	2.982484	
10	5.594943	

Name: service_time, dtype: float64

Test Set (Empty Cells):

	day	hour
0	1	0
2	1	2
5	1	5
6	1	6
8	1	8

/usr/local/lib/python3.10/dist-packages/pymc/gp/gp.py:56: FutureWarning:

The 'noise' parameter has been changed to 'sigma' in order to standardize the GP API and will be deprecated in future releases.

100.00% [2000/2000 01:19<00:00 Sampling chain 0, 1 divergences]
100.00% [2000/2000 01:25<00:00 Sampling chain 1, 0 divergences]
100.00% [2000/2000 00:34<00:00]

	day	hour	service_time
163	7	14	7.984782
164	7	15	7.473807
165	7	16	6.435730
166	7	17	5.178294
167	7	20	2.435746

We will take the predictions of our GP model including the truth data i.e. filled_grid_gp that we found in part 2 section 2, and convert the service_time column into a 7 x 24 matrix.

```
[ ] # Pivot the DataFrame to get days as rows and hours as columns
    service_time_matrix = filled_grid_gp.pivot(index='day', columns='hour', values='service_time')

    # Ensure the matrix has 7 rows (days) and 24 columns (hours)
    # Reindex to ensure all days and hours are present
    full_index = pd.Index(range(1, 8), name='day')
    full_columns = pd.Index(range(0, 24), name='hour')

    service_time_matrix = service_time_matrix.reindex(index=full_index, columns=full_columns)

    # Convert to a NumPy array
    service_time_matrix = service_time_matrix.to_numpy()

[ ] df_ = Sim_func(service_time_matrix)
    ## Converting the simulation results into a numpy array from a PD dataframe.
    preds_results = np.array(df_.iloc[:,2:]).mean(axis = 1)
```

```
[ ] from sklearn.metrics import mean_squared_error
mse = mean_squared_error(true_results, preds_results)
print(f"Mean squared error is: {mse:.4f}")
```

Mean squared error is: 0.1567

2. Plot of the average waiting as a function of time of the week (following the example from above Tuesday between 01:00 to 02:00 is the 50th hour):

```
plt.figure()
plt.plot(np.arange(167), true_results, label = 'True')
plt.plot(np.arange(167), preds_results, label = 'Predicted')

plt.legend(loc='best')
plt.xlabel('Hour of the week')
plt.ylabel('Average waiting time')
plt.title('Average waiting as a function of time of the week')

plt.legend()
plt.show()
```

