COVID-19 Mobility Insights Dashboard

Using Markov Models, HMM, and Queuing Theory

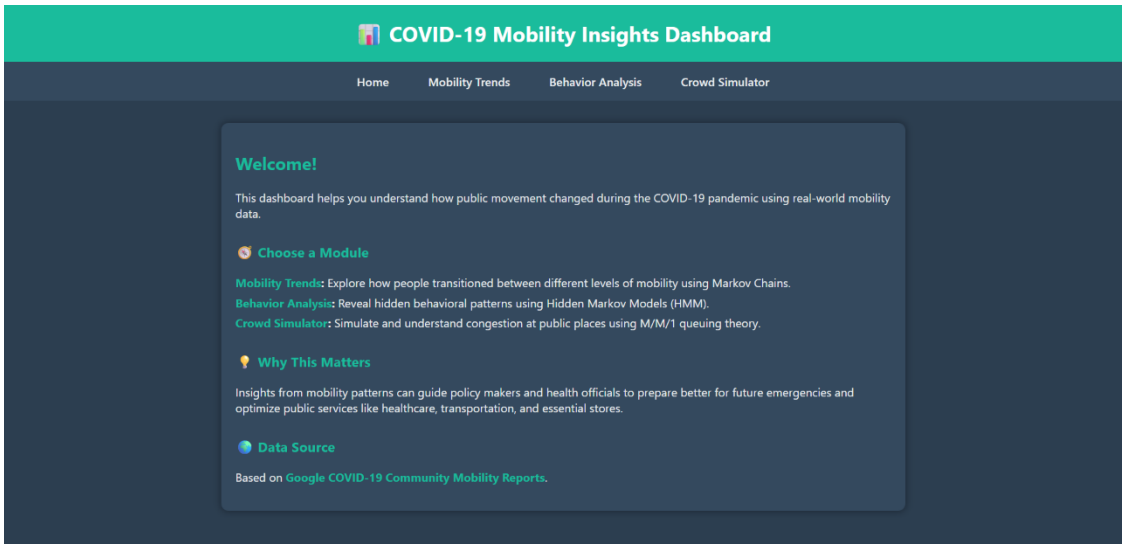
S.No	Roll Number	Name	Section
1	22F-3638	Nehal	BSSE-6A
2	22F-3707	Ibrahim Khan	BSSE-6A
3	22F-3725	Abbas Hafeez	BSSE-6A
4	22F-3630	Khushbakht Naeem	BSSE-6A

1. Detail of the Web-Based System

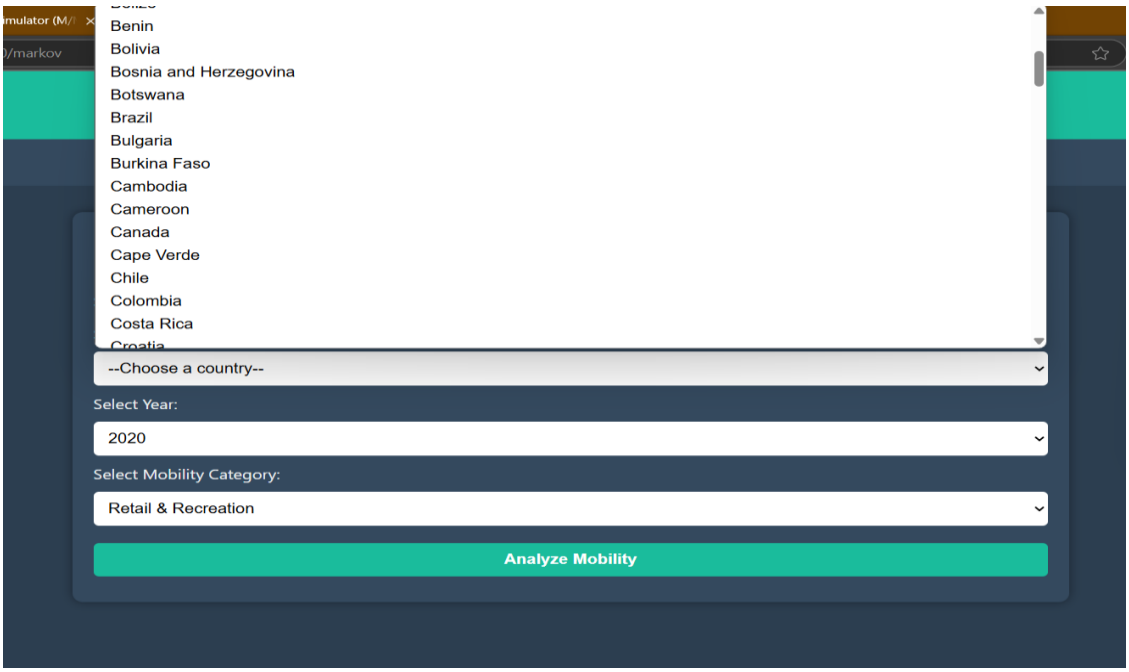
This web-based dashboard helps analyze mobility changes during the COVID-19 pandemic using real-world data. Users can:

- Select a country, year, and category to view **mobility trends**.
- Use **Markov Chains** to identify transition probabilities between mobility states.
- Apply **Hidden Markov Models** to infer unseen policy changes.
- Use **M/M/1 Queuing Theory** to simulate public congestion scenarios.
- Download **charts, CSVs, and PDF reports** for each module.
- View visualizations like pie charts, timelines, and Viterbi paths.
- Dataset: https://www.gstatic.com/covid19/mobility/Global_Mobility_Report.csv

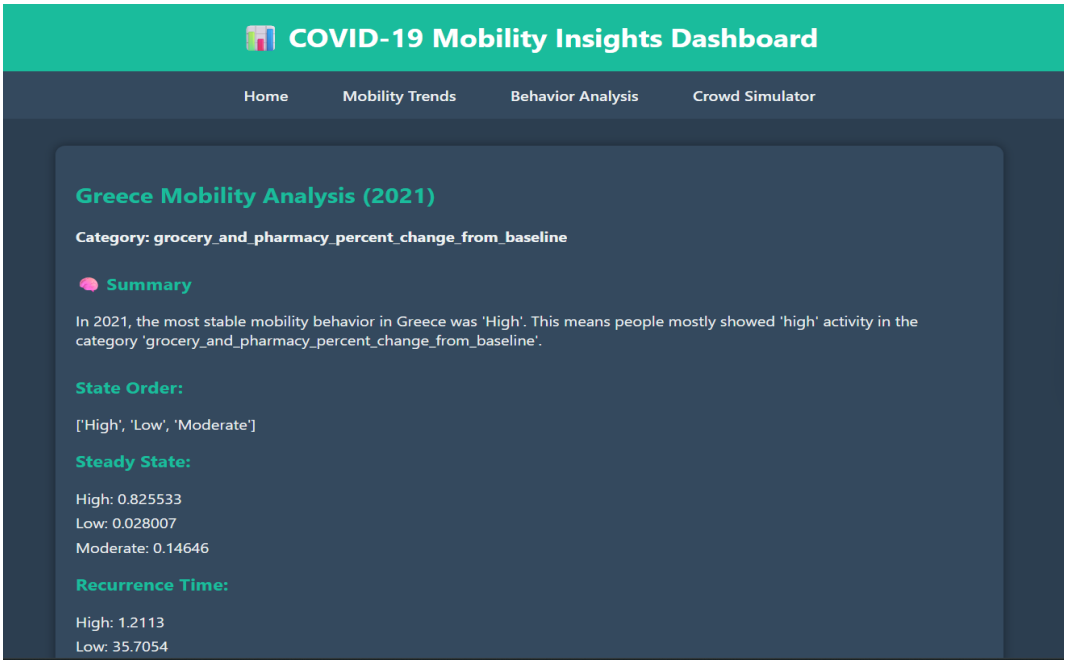
Screenshots:

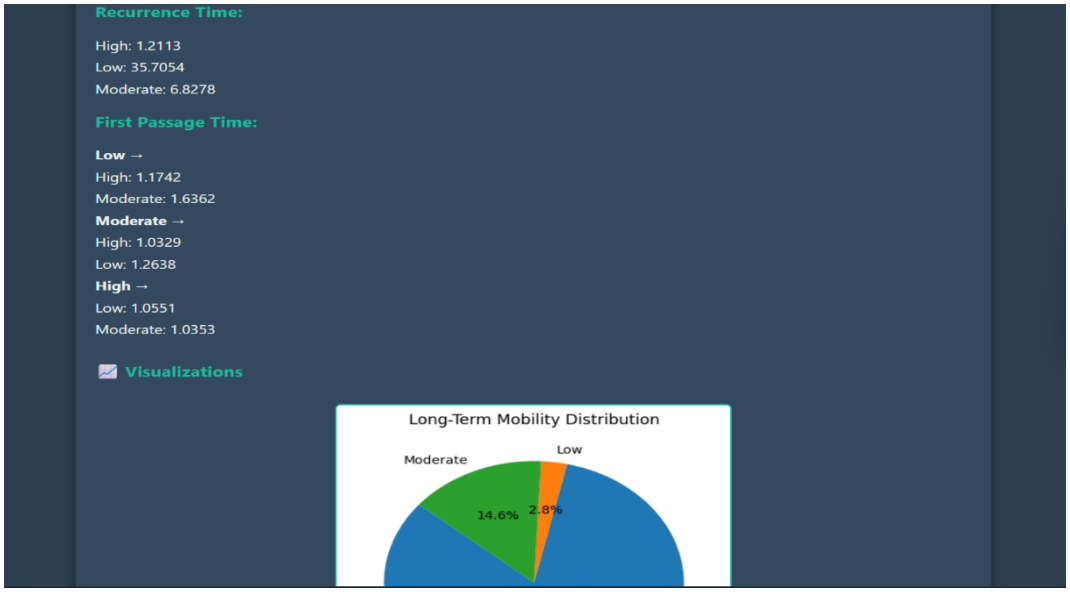


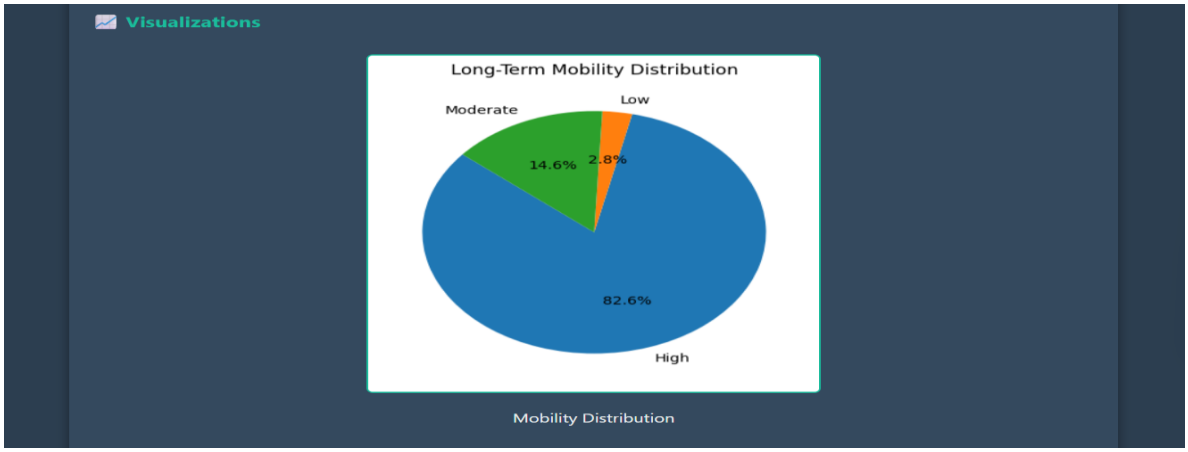
Mobility Trends Input Form:



Results:

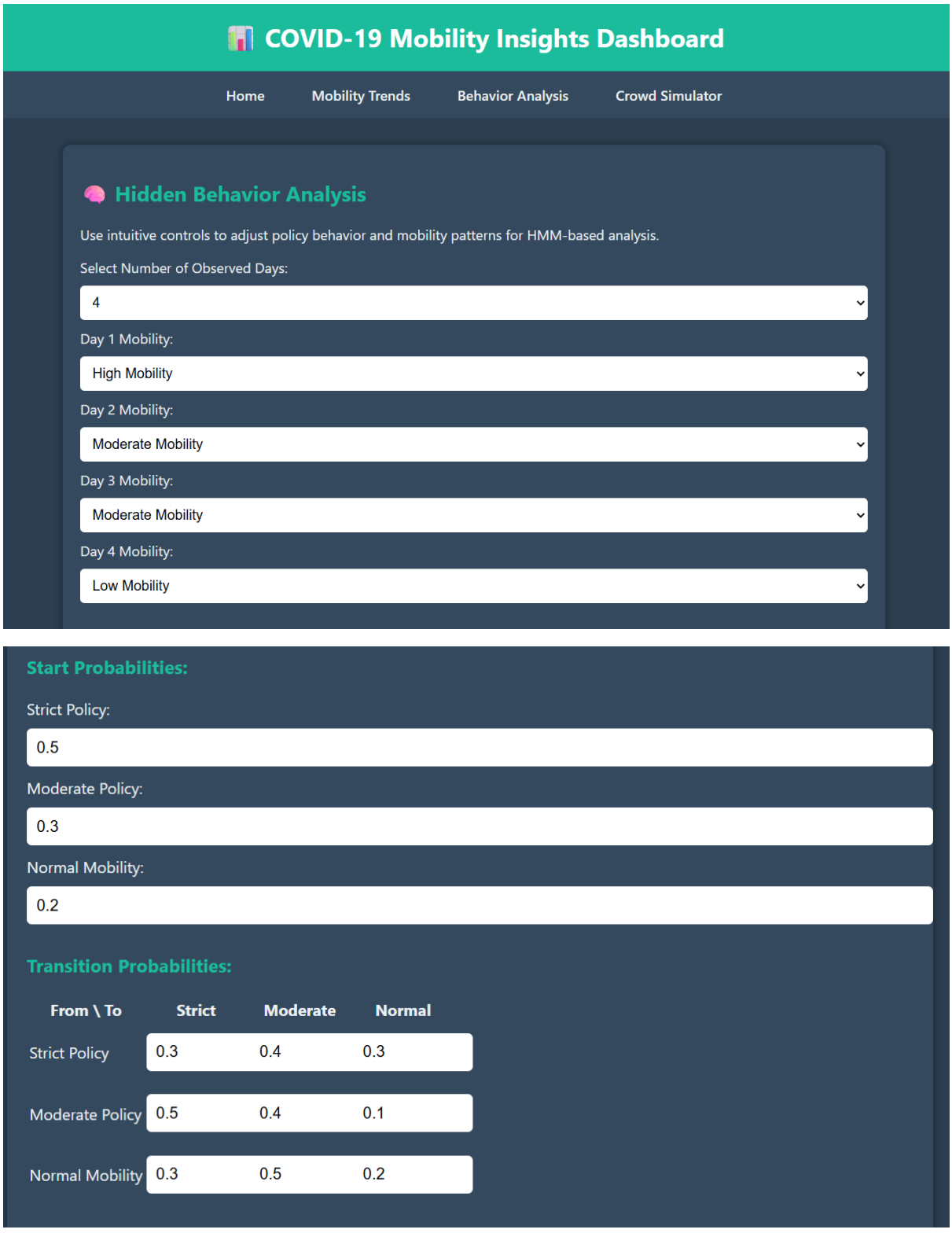

COVID-19 Mobility Insights Dashboard
Home
Mobility Trends
Behavior Analysis
Crowd Simulator
Greece Mobility Analysis (2021)
Category: grocery_and_pharmacy_percent_change_from_baseline
Summary
In 2021, the most stable mobility behavior in Greece was 'High'. This means people mostly showed 'high' activity in the category 'grocery_and_pharmacy_percent_change_from_baseline'.
State Order:
['High', 'Low', 'Moderate']
Steady State:
High: 0.825533
Low: 0.028007
Moderate: 0.14646
Recurrence Time:
High: 1.2113
Low: 35.7054


Recurrence Time:
High: 1.2113
Low: 35.7054
Moderate: 6.8278
First Passage Time:
Low →
High: 1.1742
Moderate: 1.6362
Moderate →
High: 1.0329
Low: 1.2638
High →
Low: 1.0551
Moderate: 1.0353
Visualizations
Long-Term Mobility Distribution
Moderate
Low
14.6%
2.8%



The timeline spans {{N}} days, where each day represents a valid entry in the mobility dataset for the selected country, year, and category. Due to daily-level granularity and the inclusion of multiple regions per country, the number of "days" may appear higher (e.g., 2000–3000) as each region's daily entry is considered separately in the timeline sequence.

Behavioural Analysis Input Form:



Transition Probabilities:

From \ To	Strict	Moderate	Normal
Strict Policy	0.3	0.4	0.3
Moderate Policy	0.5	0.4	0.1
Normal Mobility	0.3	0.5	0.2

Emission Probabilities:

Policy \ Observed	Low	Moderate	High
Strict Policy	0.3	0.4	0.3
Moderate Policy	0.5	0.4	0.1
Normal Mobility	0.4	0.3	0.3

Run Analysis

Forward Algorithm:

Likelihood of observed sequence: 0.016688

Most Likely Policy Sequence (Viterbi):

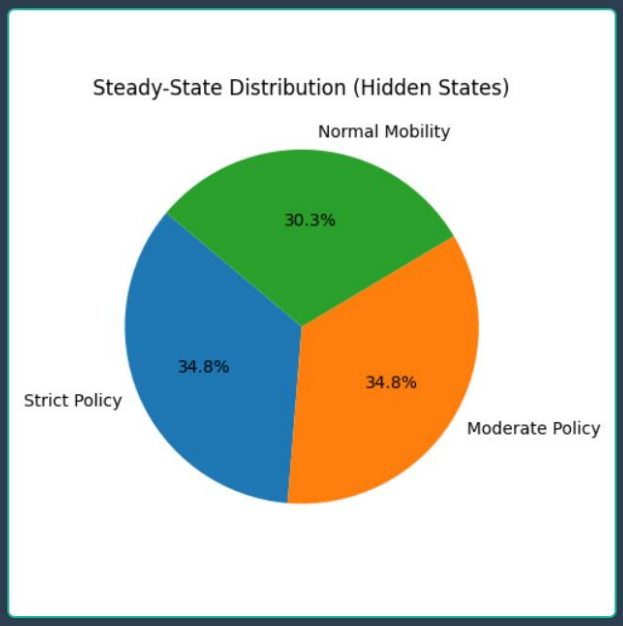
Strict Policy
Moderate Policy
Strict Policy
Moderate Policy

Steady-State Distribution:

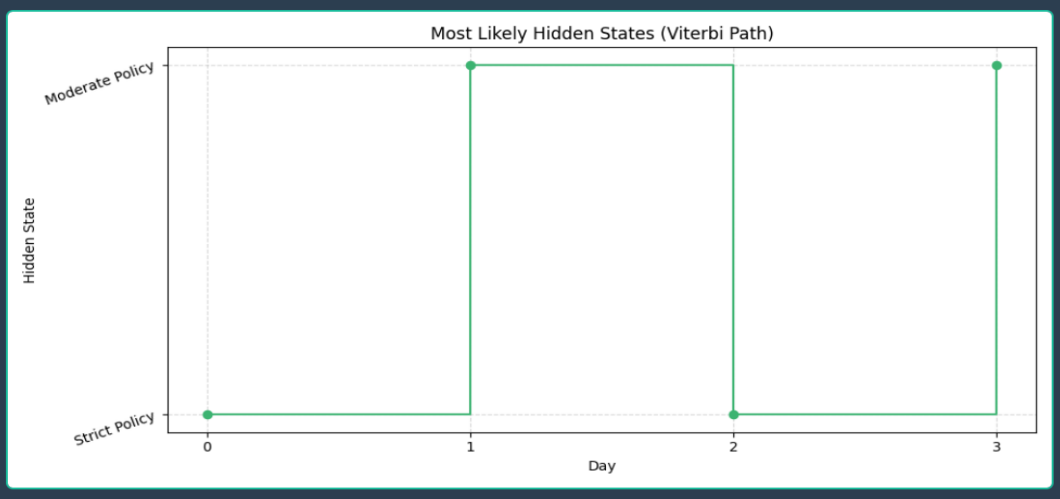
Strict Policy: 0.34827836390539857
Moderate Policy: 0.34827836390539857
Normal Mobility: 0.3034432721892029

Visualizations

Visualizations



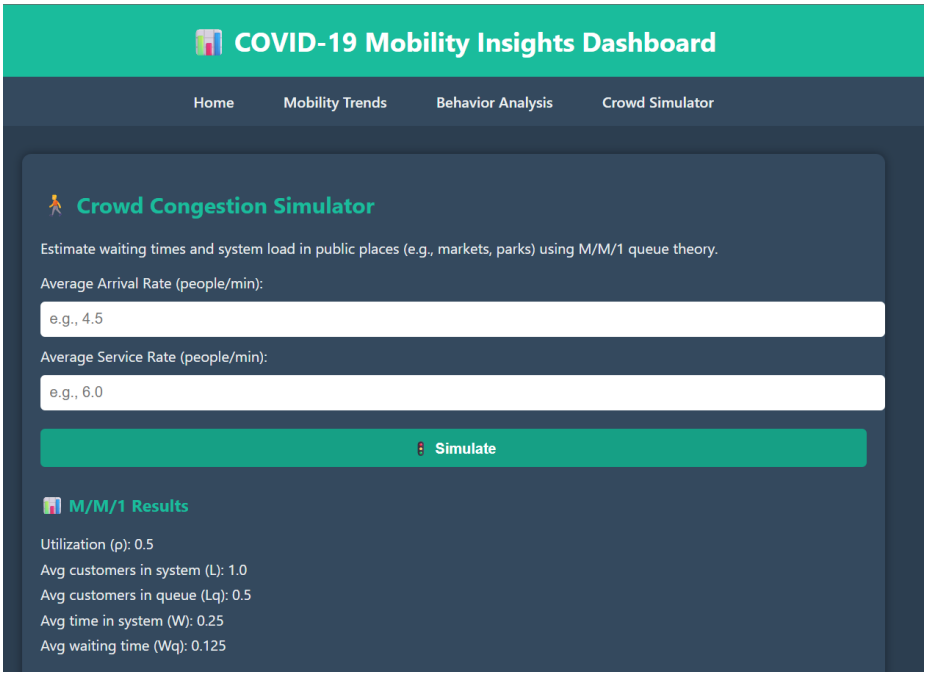
Hidden State Distribution



Most Likely Policy Path

Download PDF Download CSV

Crowd Simulator:



2. Markov Models

Problem:

We aimed to understand how people's mobility behavior transitioned daily between states like Low, Moderate, and High mobility using actual country-level data.

Solution:

Using **Markov Chains**, we built a transition matrix from mobility states and computed:

- Steady state probabilities
- Recurrence times
- First passage and absorption times

These were visualized through charts and summarized in plain language to help non-technical users understand dominant behaviors.

3. Hidden Markov Models

Problem:

We wanted to infer hidden government policy behavior (like lockdowns or re-openings) from observed mobility data.

Solution:

We implemented the **Viterbi Algorithm** and **Forward Algorithm** from scratch to:

- Compute the most probable sequence of hidden policy states
- Calculate the likelihood of observed mobility sequences

- Derive steady-state distributions of hidden states

Users can adjust transition and emission probabilities and visualize the hidden behavior using pie and line charts.

4. Queuing Theory

Problem:

We simulated how crowds form at public places like hospitals or stores under changing arrival and service rates during the pandemic.

Solution:

We used the **M/M/1 Queuing Model** to compute:

- Utilization (ρ)
- Average people in system and queue (L, Lq)
- Time spent in system and queue (W, Wq)

The module includes natural explanations and a visualization summarizing congestion metrics to help decision-makers plan better.

5. Code

Language: Python (Flask + HTML/CSS for frontend)

Folder Structure:

```
COVID_MOBILITY_SYSTEM/
├── app.py
├── requirements.txt
│
├── data/
│   └── Global_Mobility_Report.csv
│
├── modules/
│   ├── __pycache__/
│   ├── hmm_model.py
│   ├── markov_model.py
│   ├── mm1_queue.py
│   ├── preprocess.py
│   └── visuals.py
│
├── static/
│   ├── styles.css
│   └── plots/
│       ├── hidden_steady_pie.png
│       ├── hmm_report.csv
│       ├── hmm_report.pdf
│       ├── mm1_summary.png
│       ├── state_line.png
│       ├── steady_pie.png
│       ├── timeline_report.pdf
│       ├── timeline.csv
│       └── viterbi_path.png
│
├── templates/
│   ├── hmm.html
│   ├── index.html
│   ├── markov.html
│   ├── queue.html
│   └── result.html
```

Modules:

Markov_model.py:

```
from collections import defaultdict

# ------------------------------
# Build a normalized transition matrix from observed state sequence
# ------------------------------
def build_transition_matrix(states, state_order=None):
    """
```

```
    Constructs a transition probability matrix from a sequence of observed
states.

    Args:
        states (list): List of observed states (e.g., ['Low', 'High',
'Low', ...])
        state_order (list, optional): Specific order of unique states. If
None, sorted set is used.

    Returns:
        matrix (list of lists): Transition matrix where matrix[i][j] = P(j |
i)
        state_order (list): Order of states corresponding to matrix indices
    """
    if state_order is None:
        state_order = sorted(set(states))  # Automatically determine state
order

    state_idx = {s: i for i, s in enumerate(state_order)}  # Map state to
index
    n = len(state_order)
    matrix = [[0 for _ in range(n)] for _ in range(n)]  # Initialize zero
matrix

    # Count transitions from state a → b
    for a, b in zip(states[:-1], states[1:]):
        i, j = state_idx[a], state_idx[b]
        matrix[i][j] += 1

    # Normalize each row to convert to probabilities
    for i in range(n):
        row_sum = sum(matrix[i])
        if row_sum == 0:
            matrix[i][i] = 1  # Handle no transitions by self-loop
        else:
            matrix[i] = [v / row_sum for v in matrix[i]]

    return matrix, state_order

# ------------------------------
# Compute steady-state distribution using iterative method
# ------------------------------
def compute_steady_state(matrix, tol=1e-8, max_iter=1000):
    """
    Calculates the steady-state distribution π such that πP = π.

    Args:
        matrix (list of lists): Transition matrix
        tol (float): Convergence threshold
        max_iter (int): Maximum number of iterations

    Returns:
        dict: Mapping index → steady-state probability
    """
    n = len(matrix)
    prob = [1/n] * n  # Start with uniform distribution
```

```
    for _ in range(max_iter):
        new_prob = [0] * n
        for j in range(n):
            new_prob[j] = sum(prob[i] * matrix[i][j] for i in range(n))
        diff = sum(abs(new_prob[i] - prob[i]) for i in range(n))
        prob = new_prob
        if diff < tol:
            break  # Converged

    return {i: round(p, 6) for i, p in enumerate(prob)}

# ------------------------------
# Compute Mean First Recurrence Time for each state
# ------------------------------
def compute_recurrence_times(steady_probs, state_order):
    """
    Calculates average recurrence time for each state: R_i = 1 / π_i

    Args:
        steady_probs (dict): Mapping index → steady-state prob
        state_order (list): Mapping of indices to state names

    Returns:
        dict: Mapping state name → recurrence time
    """
    return {
        state_order[i]: round(1 / prob, 4)
        for i, prob in steady_probs.items()
    }

# ------------------------------
# Gaussian elimination solver for linear equations (Ax = b)
# ------------------------------
def solve_linear(A, b):
    """
    Solves Ax = b using Gaussian elimination without any external library.

    Args:
        A (list of lists): Coefficient matrix
        b (list): Constant vector

    Returns:
        list: Solution vector x
    """
    n = len(A)
    for i in range(n):
        # Pivot if diagonal element is zero
        if A[i][i] == 0:
            for j in range(i+1, n):
                if A[j][i] != 0:
                    A[i], A[j] = A[j], A[i]
                    b[i], b[j] = b[j], b[i]
                    break

        # Normalize pivot row
```

```
            factor = A[i][i]
            A[i] = [a / factor for a in A[i]]
            b[i] /= factor

            # Eliminate below
            for j in range(i+1, n):
                factor = A[j][i]
                A[j] = [a - factor * A[i][k] for k, a in enumerate(A[j])]
                b[j] -= factor * b[i]

        # Back-substitution
        x = [0] * n
        for i in reversed(range(n)):
            x[i] = b[i] - sum(A[i][j] * x[j] for j in range(i+1, n))

        return x

    # ------------------------------
    # Compute First Passage Time between states
    # ------------------------------
    def compute_first_passage(matrix, state_order):
        """
        Computes expected steps from state i to j (i ≠ j) using linear equations.

        Args:
            matrix (list of lists): Transition matrix
            state_order (list): Mapping of indices to state names

        Returns:
            dict: Nested dictionary mapping state_i → state_j → E[steps]
        """
        n = len(state_order)
        passage = {}

        for j in range(n):  # Target state
            for i in range(n):  # Start state
                if i == j:
                    continue

                A, b = [], []
                for k in range(n):
                    row = []
                    if k == j:
                        # Set equation: x_j = 0
                        row = [0] * n
                        row[k] = 1
                        b.append(0)
                    else:
                        for l in range(n):
                            if k == l:
                                row.append(1)
                            elif l == j:
                                row.append(0)
                            else:
                                row.append(-matrix[k][l])
                        b.append(1)
```

```
                A.append(row)

            x = solve_linear([r[:] for r in A], b[:])
            if state_order[i] not in passage:
                passage[state_order[i]] = {}
            passage[state_order[i]][state_order[j]] = round(x[i], 4)

    return passage

# ------------------------------
# Compute Absorption Times for transient → absorbing states
# ------------------------------
def compute_absorption(matrix, state_order):
    """
    Computes expected absorption time for transient states into absorbing
ones.
    Assumes absorbing states have P[i][i] = 1 and no outgoing transitions.

    Args:
        matrix (list of lists): Transition matrix
        state_order (list): State labels

    Returns:
        dict: State → expected absorption time (if any absorbing states exist)
    """
    n = len(state_order)
    absorbing = [i for i in range(n) if matrix[i][i] == 1.0]
    transient = [i for i in range(n) if i not in absorbing]

    if not absorbing or not transient:
        return None  # No absorbing states present

    # Build Q matrix (transient-to-transient)
    Q = [[matrix[i][j] for j in transient] for i in transient]

    # Compute I - Q
    I_minus_Q = [
        [(1 if i == j else 0) - Q[i][j] for j in range(len(Q))]
        for i in range(len(Q))
    ]

    b = [1] * len(Q)  # RHS for time equations
    t_vals = solve_linear(I_minus_Q, b)

    return {
        state_order[transient[i]]: round(t_vals[i], 4)
        for i in range(len(t_vals))
    }
```

Hmm_model.py

```
# ------------------------------
# Forward Algorithm
# ------------------------------
def forward_algorithm(obs_seq, states, start_prob, trans_prob, emit_prob):
    """
```

```
    Computes the probability of the observation sequence given the model (P(O
| λ)) using the Forward Algorithm.

    Args:
        obs_seq (list): Sequence of observed emissions (e.g., ['High',
'Moderate', 'Low'])
        states (list): List of hidden states
        start_prob (dict): Initial probabilities for each state
        trans_prob (dict of dict): Transition probability from state i to
state j
        emit_prob (dict of dict): Emission probability of observation o from
state s

    Returns:
        float: Total probability of the observation sequence
    """
    T = len(obs_seq)
    N = len(states)

    # fwd[t][i] = P(O1...Ot, Qt=state_i)
    fwd = [[0.0 for _ in range(N)] for _ in range(T)]

    # Initialization step (t = 0)
    for i in range(N):
        fwd[0][i] = start_prob[states[i]] *
emit_prob[states[i]].get(obs_seq[0], 1e-6)  # Use small default if missing

    # Recursion step
    for t in range(1, T):
        for j in range(N):
            total = 0.0
            for i in range(N):
                total += fwd[t - 1][i] * trans_prob[states[i]][states[j]]
            fwd[t][j] = total * emit_prob[states[j]].get(obs_seq[t], 1e-6)

    # Termination step: sum of final step probabilities
    return round(sum(fwd[T - 1]), 6)

# -----------------------------
# Viterbi Algorithm
# -----------------------------
def viterbi_algorithm(obs_seq, states, start_prob, trans_prob, emit_prob):
    """
    Finds the most likely sequence of hidden states given observations using
the Viterbi algorithm.

    Args:
        obs_seq (list): Observed emission sequence
        states (list): Possible hidden states
        start_prob (dict): Starting probability of each state
        trans_prob (dict of dict): State transition probabilities
        emit_prob (dict of dict): Emission probabilities for each observation
given a state

    Returns:
        list: Most probable path of hidden states
```

```
    """
    T = len(obs_seq)
    N = len(states)

    # viterbi[t][i]: max probability of any path ending in state_i at time t
    viterbi = [[0.0] * N for _ in range(T)]
    backpointer = [[0] * N for _ in range(T)]  # To trace back the path

    # Initialization (t = 0)
    for i in range(N):
        viterbi[0][i] = start_prob[states[i]] *
emit_prob[states[i]].get(obs_seq[0], 1e-6)

    # Recursion
    for t in range(1, T):
        for j in range(N):
            max_prob = 0.0
            max_index = 0
            for i in range(N):
                prob = viterbi[t - 1][i] * trans_prob[states[i]][states[j]]
                if prob > max_prob:
                    max_prob = prob
                    max_index = i
            viterbi[t][j] = max_prob * emit_prob[states[j]].get(obs_seq[t],
1e-6)
            backpointer[t][j] = max_index

    # Backtrace: find last state with max prob
    last_state = max(range(N), key=lambda i: viterbi[T - 1][i])
    best_path = [last_state]

    # Reconstruct the full path backward
    for t in range(T - 1, 0, -1):
        best_path.insert(0, backpointer[t][best_path[0]])

    return [states[i] for i in best_path]

# ------------------------------
# Compute Hidden-State Steady-State Distribution
# ------------------------------
def compute_hidden_steady_state(states, trans_prob, max_iter=1000, tol=1e-8):
    """
    Computes the steady-state distribution over hidden states using iterative
convergence.

    Args:
        states (list): List of hidden states
        trans_prob (dict of dict): Transition probabilities between states
        max_iter (int): Maximum number of iterations to run
        tol (float): Convergence threshold

    Returns:
        dict: Mapping of state name → steady-state probability
    """
    num_states = len(states)
    pi = [1.0 / num_states] * num_states  # Start with uniform distribution
```

```
    for _ in range(max_iter):
        new_pi = [0.0] * num_states
        for j in range(num_states):
            for i in range(num_states):
                new_pi[j] += pi[i] * trans_prob[states[i]][states[j]]

        # Normalize probabilities
        total = sum(new_pi)
        new_pi = [x / total for x in new_pi]

        # Check convergence
        if max(abs(new_pi[i] - pi[i]) for i in range(num_states)) < tol:
            break
        pi = new_pi

    return {states[i]: pi[i] for i in range(num_states)}
```

MM1_queue.py

```
def mm1_metrics(arrival_rate, service_rate):
    """
    Computes standard performance metrics for an M/M/1 queuing system.

    Args:
        arrival_rate (float): λ, the average arrival rate (e.g., customers per
minute)
        service_rate (float): μ, the average service rate (e.g., customers
served per minute)

    Returns:
        dict: A dictionary containing key M/M/1 queue metrics:
              - utilization (ρ)
              - average number in system (L)
              - average number in queue (Lq)
              - average time in system (W)
              - average waiting time in queue (Wq)
              - probability system is empty (P0)

    Raises:
        ValueError: If the system is unstable (arrival_rate >= service_rate)
    """

    # ⚠ Check system stability (λ must be less than μ)
    if arrival_rate >= service_rate:
        raise ValueError("System unstable: arrival rate must be less than
service rate (λ < μ).")

    # ρ (rho): Utilization factor — how busy the server is
    rho = arrival_rate / service_rate

    # L: Average number of customers in the system
    L = rho / (1 - rho)

    # Lq: Average number of customers in the queue (excluding the one being
served)
    Lq = (rho ** 2) / (1 - rho)
```

```
    # W: Average time a customer spends in the system (waiting + service)
    W = 1 / (service_rate - arrival_rate)

    # Wq: Average time a customer spends waiting in the queue
    Wq = rho / (service_rate - arrival_rate)

    # P0: Probability that there are 0 customers in the system (idle state)
    P0 = 1 - rho

    return {
        'utilization': round(rho, 4),  # Server utilization
        'L': round(L, 4),              # Avg # in system
        'Lq': round(Lq, 4),            # Avg # in queue
        'W': round(W, 4),              # Avg time in system
        'Wq': round(Wq, 4),            # Avg time in queue
        'P0': round(P0, 4)             # Idle probability
    }
```

preprocess.py

```
import pandas as pd

# ----------------------------
# Load the CSV and preprocess dates
# ----------------------------
def load_csv(csv_path):
    """
    Loads the Global Mobility CSV dataset and adds a 'year' column.

    Args:
        csv_path (str): Path to the CSV file.

    Returns:
        pd.DataFrame: Cleaned DataFrame with 'year' extracted from the 'date'
column.
    """
    df = pd.read_csv(csv_path, low_memory=False)
    df['date'] = pd.to_datetime(df['date'], errors='coerce')  # Convert to
datetime
    df['year'] = df['date'].dt.year  # Extract year for filtering
    return df

# ----------------------------
# Categorize mobility values into states
# ----------------------------
def categorize_states(series):
    """
    Categorizes numeric mobility data into qualitative labels: Low, Moderate,
High.

    Args:
        series (pd.Series): A numeric Pandas Series (e.g., % change in
mobility)

    Returns:
        pd.Series: A categorical Series with values: 'Low', 'Moderate', 'High'
    """
```

```
    bins = [-float('inf'), -20, 5, float('inf')]  # Mobility thresholds
    labels = ['Low', 'Moderate', 'High']          # Category labels
    return pd.cut(series, bins=bins, labels=labels)

# ----------------------------
# Filter by country/year and convert to mobility states
# ----------------------------
def get_mobility_states(df, country, year, category):
    """
    Extracts and categorizes mobility data into states for a specific country
and year.

    Args:
        df (pd.DataFrame): The loaded mobility DataFrame.
        country (str): Country name to filter.
        year (int): Year to filter (e.g., 2021).
        category (str): Column name for mobility type (e.g.,
'retail_and_recreation_percent_change_from_baseline').

    Returns:
        list: A list of states (Low/Moderate/High) representing daily mobility
behavior.
    """
    filtered = df[(df['country_region'] == country) & (df['year'] == year)]
    filtered = filtered.dropna(subset=[category])  # Ensure no missing data
    states = categorize_states(filtered[category]).dropna().tolist()
    return states

# ----------------------------
# Load + extract states in one call (shortcut)
# ----------------------------
def get_state_sequence(csv_path, column, country='Pakistan', year=2021):
    """
    Loads data and returns categorized mobility states in one step.

    Args:
        csv_path (str): CSV file path
        column (str): Mobility category column name
        country (str): Default is 'Pakistan'
        year (int): Default is 2021

    Returns:
        list: State sequence (Low, Moderate, High)
    """
    df = load_csv(csv_path)
    return get_mobility_states(df, country, year, column)
```

visuals.py

```
import matplotlib
matplotlib.use('Agg')  # Use non-interactive backend for Flask/production use

import matplotlib.pyplot as plt
import os

def plot_steady_pie(steady, labels, out_path):
    """
```

```
    Plots a pie chart showing steady-state probabilities of each mobility
state.

    Args:
        steady (list of float): Steady-state probabilities.
        labels (list of str): Corresponding labels for states.
        out_path (str): Path to save the pie chart image.
    """
    plt.figure(figsize=(5, 5))
    plt.pie(steady, labels=labels, autopct='%1.1f%%', startangle=140)
    plt.title("Long-Term Mobility Distribution")
    plt.savefig(out_path, bbox_inches='tight')
    plt.close()

def plot_state_timeline(sequence, save_path, csv_path=None):
    """
    Plots a timeline of observed mobility states and optionally exports as
CSV.

    Args:
        sequence (list of str): Sequence of states ('Low', 'Moderate',
'High').
        save_path (str): Path to save timeline plot image.
        csv_path (str): Optional path to save CSV of timeline.
    """
    import csv

    state_map = {"Low": 0, "Moderate": 1, "High": 2}
    y_labels = ["Low", "Moderate", "High"]
    mapped_states = [state_map.get(s, -1) for s in sequence]
    x = list(range(len(mapped_states)))

    # Downsample if too long
    MAX_POINTS = 500
    if len(x) > MAX_POINTS:
        step = len(x) // MAX_POINTS
        x = x[::step]
        mapped_states = mapped_states[::step]

    # Save to CSV if requested
    if csv_path:
        with open(csv_path, 'w', newline='') as f:
            writer = csv.writer(f)
            writer.writerow(["Day", "State"])
            reverse_map = {v: k for k, v in state_map.items()}
            for i in range(len(x)):
                writer.writerow([x[i], reverse_map.get(mapped_states[i],
'Unknown')])

    # Plotting
    plt.figure(figsize=(12, 4))
    plt.plot(x, mapped_states, color='royalblue', linewidth=1.5, alpha=0.85)

    # Vertical lines at state changes
    for i in range(1, len(mapped_states)):
        if mapped_states[i] != mapped_states[i - 1]:
```

```
            plt.axvline(x[i], color='gray', linestyle=':', alpha=0.2)

    plt.yticks([0, 1, 2], y_labels)
    plt.xlabel("Day")
    plt.ylabel("Mobility State")
    plt.title("📊 Mobility State Timeline")
    plt.grid(True, axis='y', linestyle='--', alpha=0.4)
    plt.tight_layout()
    plt.savefig(save_path)
    plt.close()

def plot_viterbi_path(states, out_path):
    """
    Plots the most likely hidden states using Viterbi algorithm.

    Args:
        states (list of str): Viterbi-decoded sequence of hidden states.
        out_path (str): Path to save the line chart.
    """
    plt.figure(figsize=(10, 5))
    plt.plot(range(len(states)), states, drawstyle='steps-post', marker='o',
color='mediumseagreen')
    plt.title("Most Likely Hidden States (Viterbi Path)")
    plt.xlabel("Day")
    plt.ylabel("Hidden State")
    plt.xticks(range(len(states)))
    plt.yticks(sorted(set(states)))
    plt.gca().set_yticklabels(sorted(set(states)), rotation=20)
    plt.grid(True, linestyle='--', alpha=0.5)
    plt.tight_layout()
    plt.savefig(out_path)
    plt.close()

def plot_hidden_steady_pie(steady_dict, out_path):
    """
    Plots a pie chart of steady-state probabilities from hidden states.

    Args:
        steady_dict (dict): State → probability mapping.
        out_path (str): Path to save chart image.
    """
    labels = list(steady_dict.keys())
    sizes = list(steady_dict.values())

    plt.figure(figsize=(5, 5))
    plt.pie(sizes, labels=labels, autopct='%1.1f%%', startangle=140)
    plt.title("Steady-State Distribution (Hidden States)")
    plt.tight_layout()
    plt.savefig(out_path)
    plt.close()

def plot_mm1_summary(metrics, out_path):
    """
    Plots bar chart for key M/M/1 Queueing metrics.

    Args:
```

```
        metrics (dict): Dictionary with keys 'L', 'Lq', 'W', 'Wq'.
        out_path (str): Path to save image.
    """
    labels = ['Avg Customers (L)', 'Avg in Queue (Lq)', 'Avg Time in System
(W)', 'Avg Wait Time (Wq)']
    values = [metrics['L'], metrics['Lq'], metrics['W'], metrics['Wq']]

    plt.figure(figsize=(8, 4))
    plt.bar(labels, values, color='teal')
    plt.title('M/M/1 Queueing Summary')
    plt.tight_layout()
    plt.savefig(out_path)
    plt.close()
```

app.py

```
import os
from flask import Flask, render_template, request, render_template_string,
url_for, send_file
from modules.preprocess import load_csv, get_mobility_states
from modules.visuals import plot_steady_pie, plot_state_timeline,
plot_mm1_summary
from modules.markov_model import (
    build_transition_matrix,
    compute_steady_state,
    compute_recurrence_times,
    compute_first_passage,
    compute_absorption
)
from modules.hmm_model import forward_algorithm, viterbi_algorithm,
compute_hidden_steady_state
from modules.mm1_queue import mm1_metrics
import csv
from fpdf import FPDF

app = Flask(__name__)

#  ◇  Load Global Data on Startup
DATA_PATH = 'data/Global_Mobility_Report.csv'
GLOBAL_DATA = load_csv(DATA_PATH)

#  ◇  Home Route
@app.route('/')
def index():
    return render_template('index.html')


#  ◇  Markov Model Route
@app.route('/markov', methods=['GET', 'POST'])
def markov():
    if request.method == 'POST':
        # Step 1: Get user input
        country = request.form['country']
        year = int(request.form['year'])
        category = request.form['category']

        try:
```

```
            sequence = get_mobility_states(GLOBAL_DATA, country, year,
category)
        except Exception as e:
            return render_template('markov.html', error=str(e))

        # Step 2: Run Markov Model computations
        matrix, order = build_transition_matrix(sequence)
        steady_raw = compute_steady_state(matrix)
        steady = {order[i]: steady_raw[i] for i in range(len(order))}
        recurrence = compute_recurrence_times(steady_raw, order)
        passage = compute_first_passage(matrix, order)
        absorption = compute_absorption(matrix, order)

        # Step 3: Save charts & timeline
        img_dir = 'static/plots'
        os.makedirs(img_dir, exist_ok=True)
        pie_path = os.path.join(img_dir, 'steady_pie.png')
        line_path = os.path.join(img_dir, 'state_line.png')
        plot_steady_pie(list(steady.values()), list(steady.keys()), pie_path)

        # Step 4: Summary generation
        dominant_state = max(steady, key=steady.get)
        summary = f"In {year}, the most stable mobility behavior in {country}
was '{dominant_state}'. " \
                  f"This means people mostly showed '{dominant_state.lower()}'
activity in the category '{category}'."

        # Step 5: Save timeline chart + CSV + PDF
        timeline_csv = os.path.join(img_dir, 'timeline.csv')
        timeline_pdf = os.path.join(img_dir, 'timeline_report.pdf')
        plot_state_timeline(sequence, line_path, csv_path=timeline_csv)

        # Export timeline summary as PDF
        pdf = FPDF()
        pdf.add_page()
        pdf.set_font("Arial", size=12)
        pdf.cell(200, 10, txt="Mobility Timeline Report", ln=True, align='C')
        pdf.ln(10)
        pdf.multi_cell(0, 8, txt=summary)
        pdf.image(line_path, x=10, y=pdf.get_y() + 5, w=190)
        pdf.output(timeline_pdf)

        # Step 6: Render result
        html_block = render_template_string("""
            <h3>State Order:</h3>
            <p>{{ order }}</p>

            <h3>Steady State:</h3>
            <ul>{% for k, v in steady.items() %}<li>{{ k }}: {{ v }}</li>{%
endfor %}</ul>

            <h3>Recurrence Time:</h3>
            <ul>{% for k, v in recurrence.items() %}<li>{{ k }}:
{{ v }}</li>{% endfor %}</ul>

            <h3>First Passage Time:</h3>
```

```
            <ul>
            {% for i, row in passage.items() %}
                <li><strong>{{ i }} →</strong>
                    <ul>
                    {% for j, val in row.items() %}
                        <li>{{ j }}: {{ val }}</li>
                    {% endfor %}
                    </ul>
                </li>
            {% endfor %}
            </ul>

            {% if absorption %}
            <h3>Absorption Times:</h3>
            <ul>
            {% for k, v in absorption.items() %}
                <li>{{ k }}: {{ v }}</li>
            {% endfor %}
            </ul>
            {% endif %}
        """, order=order, steady=steady, recurrence=recurrence,
passage=passage, absorption=absorption)

        return render_template("result.html",
                               title=f"{country} Mobility Analysis ({year})",
                               subtitle=f"Category: {category}",
                               content=html_block,
                               back_url=url_for('markov'),
                               pie_chart_url='/' + pie_path,
                               line_chart_url='/' + line_path,
                               summary_text=summary)

    return render_template('markov.html')


# ◇ Hidden Markov Model Route
@app.route('/hmm', methods=['GET', 'POST'])
def hmm():
    default_states = ['Strict Policy', 'Moderate Policy', 'Normal Mobility']

    if request.method == 'POST':
        # Step 1: Parse inputs
        obs_str = request.form.get('obs_seq', '')
        obs_seq = [x.strip() for x in obs_str.split(',') if x.strip()]

        # Step 2: Extract start, transition, and emission probabilities
        start_prob = {
            'Strict Policy': float(request.form.get('start_prob[Strict
Policy]', 0.5)),
            'Moderate Policy': float(request.form.get('start_prob[Moderate
Policy]', 0.3)),
            'Normal Mobility': float(request.form.get('start_prob[Normal
Mobility]', 0.2))
        }

        default_trans = {}
```

```
        default_emit = {}
        for state in default_states:
            default_trans[state] = {}
            default_emit[state] = {}
            for to_state in default_states:
                default_trans[state][to_state] =
float(request.form.get(f'trans[{state}][{to_state}]', 0.33))
            for obs in ['Low Mobility', 'Moderate Mobility', 'High Mobility']:
                default_emit[state][obs] =
float(request.form.get(f'emit[{state}][{obs}]', 0.33))

        # Step 3: Run HMM algorithms
        forward_prob = forward_algorithm(obs_seq, default_states, start_prob,
default_trans, default_emit)
        viterbi_path = viterbi_algorithm(obs_seq, default_states, start_prob,
default_trans, default_emit)
        steady_hidden = compute_hidden_steady_state(default_states,
default_trans)

        # Step 4: Save charts
        img_dir = 'static/plots'
        os.makedirs(img_dir, exist_ok=True)
        viterbi_path_img = os.path.join(img_dir, 'viterbi_path.png')
        steady_pie_img = os.path.join(img_dir, 'hidden_steady_pie.png')

        from modules.visuals import plot_viterbi_path, plot_hidden_steady_pie
        plot_viterbi_path(viterbi_path, viterbi_path_img)
        plot_hidden_steady_pie(steady_hidden, steady_pie_img)

        # Store for download use
        app.config["last_viterbi_path"] = viterbi_path
        app.config["last_steady_hidden"] = steady_hidden

        return render_template('hmm.html',
                               forward_prob=round(forward_prob, 6),
                               viterbi_path=viterbi_path,
                               steady_hidden=steady_hidden,
                               selected_obs=obs_seq,
                               viterbi_chart_url='/' + viterbi_path_img,
                               steady_chart_url='/' + steady_pie_img)

    return render_template('hmm.html',
                           forward_prob=None,
                           viterbi_path=[],
                           steady_hidden={},
                           selected_obs=[])


# ◇ HMM Report Downloads
@app.route('/hmm/download/pdf')
def download_hmm_pdf():
    pdf = FPDF()
    pdf.add_page()
    pdf.set_font("Arial", size=12)
    pdf.set_title("HMM Report")
```

```
    pdf.cell(200, 10, txt="COVID-19 Mobility - HMM Report", ln=True,
align='C')
    pdf.ln(10)

    pdf.cell(200, 10, txt="Most Likely Policy Path (Viterbi):", ln=True)
    for i, state in enumerate(app.config.get("last_viterbi_path", []),
start=1):
        pdf.cell(200, 10, txt=f"Day {i}: {state}", ln=True)

    pdf.ln(5)
    pdf.cell(200, 10, txt="Steady-State Distribution:", ln=True)
    for state, prob in app.config.get("last_steady_hidden", {}).items():
        pdf.cell(200, 10, txt=f"{state}: {prob:.4f}", ln=True)

    if os.path.exists("static/plots/viterbi_path.png"):
        pdf.image("static/plots/viterbi_path.png", x=10, y=pdf.get_y() + 10,
w=90)
    if os.path.exists("static/plots/hidden_steady_pie.png"):
        pdf.image("static/plots/hidden_steady_pie.png", x=110, y=pdf.get_y(),
w=90)

    pdf.output("static/plots/hmm_report.pdf")
    return send_file("static/plots/hmm_report.pdf", as_attachment=True)


@app.route('/hmm/download/csv')
def download_hmm_csv():
    filepath = "static/plots/hmm_report.csv"
    with open(filepath, mode='w', newline='') as file:
        writer = csv.writer(file)
        writer.writerow(["Day", "Viterbi State"])
        for i, state in enumerate(app.config.get("last_viterbi_path", []),
start=1):
            writer.writerow([i, state])

        writer.writerow([])
        writer.writerow(["State", "Steady Probability"])
        for state, prob in app.config.get("last_steady_hidden", {}).items():
            writer.writerow([state, prob])

    return send_file(filepath, as_attachment=True)


# ◇ Queueing Theory Route (M/M/1)
@app.route('/queue', methods=['GET', 'POST'])
def queue():
    if request.method == 'POST':
        try:
            arrival = float(request.form['arrival'])
            service = float(request.form['service'])
            result = mm1_metrics(arrival, service)

            # Plot queue summary
            plot_path = 'static/plots/mm1_summary.png'
            os.makedirs(os.path.dirname(plot_path), exist_ok=True)
            plot_mm1_summary(result, plot_path)
```

```
            # Generate summary text
            rho = result['utilization']
            summary = (
                f"The system utilization is {rho:.2f} "
                f"(server is busy {rho*100:.1f}% of the time). "
                f"On average, {result['L']:.2f} customers are in the system, "
                f"{result['Lq']:.2f} in queue. "
                f"Expected time in system is {result['W']:.2f}, "
                f"of which {result['Wq']:.2f} is spent waiting."
            )

            return render_template('queue.html',
                                   metrics=result,
                                   summary=summary,
                                   chart_url='/' + plot_path)
        except Exception as e:
            return render_template('queue.html', error=str(e))

    return render_template('queue.html')


# ◇ Run App
if __name__ == '__main__':
    app.run(debug=True)
```

static/styles.css and templates are the basic css and html codes.

6. Conclusion

This dashboard successfully translates real-world mobility data into actionable insights using Markov Chains, Hidden Markov Models, and Queuing Theory. Through Markov analysis, we identified dominant mobility behaviors and their long-term stability. The HMM module uncovered hidden policy patterns driving observed movement levels, while the M/M/1 queuing model quantified potential congestion levels at public places. Together, these models not only visualize how public mobility evolved during the COVID-19 pandemic but also provide predictive tools to support smarter, data-driven decisions for future health emergencies and urban planning.