```
% ChadEdJasProChris
% ENGS 27; Final Project - Matlab
% November 18, 2020

%NOTE: BEFORE RUNNING THIS SCRIPT YOU MUST FIRST RUN
 "model_informing.m"

% Code description: This code computes the transition model for every
% reasonable cell in the matrix data presented to it
% A reasonable cell here is defined as a cell that at one point in
 time
% has data in it

% The code is generic enough that it is able to run on both actual and
% simulated data (generated randomly) perfectly

% User advice:
% To indicate the type of data to run the code on (whether simulated
 or actual)
% change the value for the parameter "REAL_DATA" below accordingly
% (1 of 2 values, 0 and 1, can be used)
% 0 tells the code to expect simulated data, which it generates at the
% start of the code on its own
% 1 tells it to expect real data. Here the assumption is that the user
 will
% provide their own data. If no data is presented, the code will run
 into an error

% Indicate the type of data type to use
% 1: real data, 0: randomly generated proxy data used to test code


REAL_DATA = 1;

% Main Code.........................................
% Boiler plate functions are after main

% Get the data to operate on
% If no data (that is 0 is indicated as the value of "Real_Data"),
% use the simulated data function to create one on the fly
if REAL_DATA
    start = 3; % which day to start on
    T = 164; % how many days to include in the model
    EXCLUDED_CELLS = [6, 7, 13, 14, 21, 35, 36, 46, 47, 48, 49];

    % geo_script_draft.m; % build the matrix to be used
    grid_size = length(protests_per_region(:, :, 1));
    num_cells = grid_size * grid_size;
    steps = protests_per_region(:, :, start:T+start);

else % If we are using simulated data follow this process
    num_cells = 25; % number of regions being modeled
```

```
    grid_size = sqrt(num_cells);
    T = 24; % how many time steps for the simulated data (Assume a
 monthly time step)
    steps = create_simulated_data(grid_size, Y*T);
end

P = zeros(2, 2, num_cells); % stores the transition matrices for the
 regions

% Compute the transition matrix for every cell/location/region
for i = 1:grid_size
    for j = 1:grid_size
        cell_number = 7*(j-1) + i;
        if ~ismember(cell_number, EXCLUDED_CELLS)
            P(:, :, cell_number) = compute_trans_M(i, j, steps,
 grid_size, T);
        end
    end
end

SUBPLOT_ROWS = 6;
SUBPLOT_COLS = 5;

% Visualization of the data matrix
labels = {'1','2', '3', '4', '5', '6', '7'};

figure_count = 3;
for count = 1:T
    index = mod(count, SUBPLOT_ROWS * SUBPLOT_COLS);
    if index == 1
        figure(figure_count);
        figure_count = figure_count + 1;

    elseif index == 0
        index = SUBPLOT_ROWS * SUBPLOT_COLS;
    end

    subplot(SUBPLOT_ROWS, SUBPLOT_COLS, index);
    h = heatmap(labels, labels, steps(:, :, count));
    colormap autumn
    h.Title = "Day: " + count;
end

% Visualization of the transition matrices
figure(figure_count);
labels = {'0','1'};
for count = 1:num_cells
    cell_number = subplot_index_to_matrix_index(count);
    if ~ismember(cell_number, EXCLUDED_CELLS)
        subplot(grid_size, grid_size, count);
        h = heatmap(labels, labels, P(:, :, cell_number));
        colormap autumn
        caxis([0 1])
        h.Title = "Region: " + count;
```

```
    end
end

% Boiler plate functions.................................

% The function provides simulated data
function steps = create_simulated_data(grid_size, T)
    steps = zeros(grid_size, grid_size, T);  % stores the data for
 each time step
    for t = 1:T
        % time step matrix representing robbery state for each
        % region (0 means no robbery, 1 means robbery)
        % generate a matrix of random binary values as ur proxy data
        steps(:, :, t) = randi([0 1], grid_size, grid_size);
        % figure(n)
        % spy(steps(:,:,n))?
    end
end

% The function brdiges all the different neighborhod state functions
 to
% allow for the use of a single function in the compute_trans_M
 function
function state = neighborhood_state(i, j, steps, t, grid_size)
    if i == 1
        if j == 1 % Left top corner: m_r = 1 and m_c = 1
            state = corner_neighborhood_state(i, j, steps, t, 1, 1);
        elseif j == grid_size % Right top corner: m_r = 1 and m_c = -1
            state = corner_neighborhood_state(i, j, steps, t, 1, -1);
        else % Top boundary: m_r = 1
            state = row_generic_neighborhood_state(i, j, steps, t, 1);
        end

    elseif i == grid_size
        if j == 1 % Left bottom corner: m_r = -1 and m_c = 1
            state = corner_neighborhood_state(i, j, steps, t, -1, 1);
        elseif j == grid_size % Right bottom corner: m_r = -1 and m_c
 = -1
            state = corner_neighborhood_state(i, j, steps, t, -1, -1);
        else % Bottom boundary: m_r = -1
            state = row_generic_neighborhood_state(i, j, steps, t,
 -1);
        end

    else
        if j == 1 % Left boundary: m_c = 1
            state = col_generic_neighborhood_state(i, j, steps, t, 1);
        elseif j == grid_size % Right boundary: m_c = -1
            state = col_generic_neighborhood_state(i, j, steps, t,
 -1);
        else % Generic case
            state = generic_neighborhood_state(i, j, steps, t);
        end
```

```
    end
end

% The function finds the neighborhood sate of a corner location
% m_r: row to find s's neighbors aside from s's row (-1: go down by 1;
 +1: go up by 1)
% m_c: col to find s's neighbors aside from s's col (-1: go left by 1;
 +1: right up by 1)
function state = corner_neighborhood_state(i, j, steps, t, m_r, m_c)
    state = max([steps(i+m_r, j, t) steps(i+m_r, j+m_c, t) steps(i, j
+m_c, t)]);
end

% The function finds the neighborhood sate of a generic location along
 a
% given row boundary
% m_r: row to find s's neighbors aside from s's row (-1: go down by 1;
 +1: go up by 1)
function state = row_generic_neighborhood_state(i, j, steps, t, m_r)
    state = max([steps(i+m_r, j-1, t) steps(i+m_r, j, t) steps(i+m_r,
 j+1, t) steps(i, j-1, t) steps(i, j, t) steps(i, j+1, t)]);
end

% The function finds the neighborhood sate of a generic location along
 a
% given column boundary
% m_c: col to find s's neighbors aside from s's col (-1: go left by 1;
 +1: right up by 1)
function state = col_generic_neighborhood_state(i, j, steps, t, m_c)
    state = max([steps(i-1, j, t) steps(i-1, j+m_c, t) steps(i, j, t)
 steps(i, j+m_c, t) steps(i+1, j, t) steps(i+1, j+m_c, t)]);
end

% The function finds the neighborhood sate of a generic location
% (one that is not a boundary case or a corner)
function state = generic_neighborhood_state(i, j, steps, t)
    state = max([steps(i-1, j-1, t) steps(i-1, j, t) steps(i-1, j+1,
 t) steps(i, j-1, t) steps(i, j, t) steps(i, j+1, t) steps(i+1, j-1,
 t) steps(i+1, j, t) steps(i+1, j+1, t)]);
end

% The function computes the probability with which some location (i,
 j)
% transitions from a state given by its neighborhood state to
% every possible state, including that of its neighborhood state in
% a time step
function trans_M = compute_trans_M(i, j, steps, grid_size, T)
    trans_M = zeros(2);
    for s = 0:1
        event_space = 0;
        sample_space = 0;
        for t = 1:T-1  % Go all the way up to the last but one time
 step
            if neighborhood_state(i, j, steps, t, grid_size) == s
```

```
                if steps(i, j, t+1) == 0
                    event_space = event_space + 1; % increase the
 event space count
                end
                sample_space = sample_space + 1; % increase the sample
 space count regardless
            end
        end
        prob_to_state_zero = 0;
        if sample_space % make sure not to divide by zero
            prob_to_state_zero = event_space/sample_space;
        end
        trans_M(s+1, 1) = prob_to_state_zero;
        trans_M(s+1, 2) = 1 - prob_to_state_zero;

    end
end


function cell_number = subplot_index_to_matrix_index(plot_index)
    i = ceil(plot_index/7);
    j = mod(plot_index, 7);
    if j == 0
        j = 7;
    end
    cell_number = 7*(j-1) + i;
end
```

Published with MATLAB® R2019b