Analysis of Cognitive Vulnerabilities Model in Python

Peter S. Hovmand and Robinson Salazar Rua

February 2, 2024

1 Install the packages

This step involves setting up the Python environment with necessary libraries like `pandas`, `numpy`, `matplotlib`, `seaborn` and using `pip`. It's crucial for accessing different tools and functions for data manipulation and visualization.

[1]:
```
%%capture
!pip install pandas
!pip install numpy
!pip install matplotlib
!pip install seaborn
```

2 Load the libraries

This segment includes imports of several key Python libraries, each serving distinct purposes:

- **`os`**: Used for operating system interactions, such as directory manipulation and path management.
- **`time`**: Facilitates time-related tasks like pausing execution and recording timestamps.
- **`subprocess`**: Enables the script to run external programs and interact with their I/O.
- **`zipfile`**: Provides functionality for reading and writing ZIP files.
- **`pandas`**: Essential for data manipulation and analysis, particularly with DataFrames.
- **`numpy`**: A cornerstone for scientific computing, supporting large arrays and matrices.
- **`matplotlib.pyplot`**: Offers a MATLAB-like plotting framework for creating various types of visualizations.
- **`seaborn`**: Builds on Matplotlib to create attractive and informative graphs.

Together, these libraries equip the script for a broad spectrum of tasks, ranging from file management to complex data analysis and visual representation.

[2]:
```
import os
import time
import subprocess
import zipfile
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

3 Record the start time

In this code snippet, the script records the current time at a specific moment using Python's time module. This is achieved by calling time.time(), which returns the current time in seconds. The returned value is stored in the variable start_time. This technique is commonly used to mark the start of a process or operation whose duration needs to be measured. By capturing the start time, the script sets a reference point from which it can calculate elapsed time later in the execution, typically for performance monitoring or timing the duration of code segments. This is a simple yet effective method for time tracking within a script.

[3]:
```
start_time = time.time()
```

4 Set the working directory

This part focuses on changing the current working directory of the Python environment to a specified path, followed by retrieving and displaying the current working directory. Initially, it uses os.chdir() to set the working directory to a specific folder on the user's desktop. This is particularly useful for aligning file operations with a desired file path, ensuring that subsequent file read or write operations occur in the correct directory. After changing the directory, the script then calls os.getcwd(), which returns the current working directory. This step is typically used to confirm that the directory change was successful. This approach streamlines file management in Python scripts, especially when working with multiple files or when a specific directory structure is required.

[4]:
```
os.chdir("C:/Users/rss188/Desktop/Python_CognitiveVulnerabilities")
os.getcwd()
```

[4]:
```
'C:\\Users\\rss188\\Desktop\\Python_CognitiveVulnerabilities'
```

5 Unzip the model

Here, the script is designed to handle the extraction of a ZIP file's contents into a specified directory. It begins by defining the path to the ZIP file, which in this case, is located within a specific folder on the user's desktop. The script then specifies the target directory where the contents of the ZIP file are to be extracted. To ensure the smooth execution of the extraction process, the script includes a command to create the target directory if it does not already exist, thereby preventing any potential directory-related errors.

Following the setup, the script employs Python's zipfile module to open and read the contents of the ZIP file. It then proceeds to extract all files and folders contained within the ZIP file into the previously specified target directory. This action is encapsulated within a context manager (with statement), ensuring that the ZIP file is properly handled and closed after the extraction process is completed. The script concludes with a print statement, confirming the successful extraction of the ZIP file's contents to the designated folder. This section exemplifies a straightforward and effective method for programmatically extracting compressed files in Python, which is particularly useful for managing file distributions and setups in software project.

[5]:
```
# Path to the zip file
```

```
zip_file_path = 'C:/Users/rss188/Desktop/Python_CognitiveVulnerabilities/Model.
 ↪zip'

# Directory where you want to extract the contents
extract_to_folder = 'C:/Users/rss188/Desktop/Python_CognitiveVulnerabilities' ␣
 ↪# Replace with your desired path

# Create the directory if it doesn't exist
os.makedirs(extract_to_folder, exist_ok=True)

# Unzip the file
with zipfile.ZipFile(zip_file_path, 'r') as zip_ref:
    zip_ref.extractall(extract_to_folder)

print(f"Extracted contents of {zip_file_path} to {extract_to_folder}")
```

```
Extracted contents of
C:/Users/rss188/Desktop/Python_CognitiveVulnerabilities/Model.zip to
C:/Users/rss188/Desktop/Python_CognitiveVulnerabilities
```

6 Create empty files for importing/exporting data to the Stella model

In this segment, the Python script is focused on initializing the environment for a data processing operation. It achieves this by creating two empty CSV files: one for importing parameters and another for exporting results. This is done using the pandas library, which is a powerful tool for data manipulation in Python. The script first creates an empty pandas DataFrame and then immediately exports it to a designated path as 'Parms.csv', intended to store parameters for subsequent processes. Similarly, another empty DataFrame is created and saved as 'Results.csv', which is reserved for storing the outcomes of those processes. This approach is a clear and efficient way to set up a structured framework for data input and output, ensuring that the necessary files are in place before starting a data processing task..

Important: The Stella model was previously configured to recognize certain file names, so it's essential to use these exact file names.

[6]:
```
# Create two empty files for importing and exporting data
pd.DataFrame().to_csv("C:/Users/rss188/Desktop/Python_CognitiveVulnerabilities/
 ↪Model/Parms.csv")
pd.DataFrame().to_csv("C:/Users/rss188/Desktop/Python_CognitiveVulnerabilities/
 ↪Model/Results.csv")
```

7 Read the model and export the results

This part is dedicated to running a baserun simulation using Stella, a software application for system dynamics modeling. The script starts by specifying the path to the Stella Architect executable. It then executes a subprocess command to run a Stella model simulation, pointing to a specific model

file located in the user's directory. Following the completion of the simulation, the script proceeds to export the results. This is accomplished by reading data from a CSV file, which contains the output of the simulation, into a pandas DataFrame named 'baserun'. This process exemplifies a straightforward and efficient method for executing system dynamics simulations and retrieving their results for further analysis or processing.

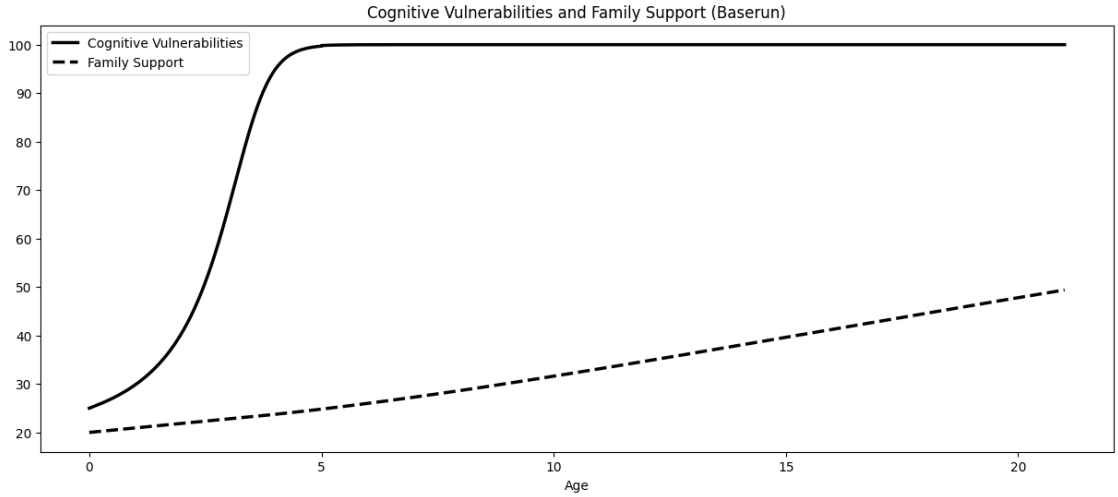
[7]:
```
# Run a baserun simulation
stella_simulator_path = "C:/Program Files/isee systems/Stella Architect/Stella␣
 ↪Architect.exe"
subprocess.run([stella_simulator_path, "-r", "C:/Users/rss188/Desktop/
 ↪Python_CognitiveVulnerabilities/Model/Developmental-Transitions-2-2-10.
 ↪stmx"])

# Export results
baserun = pd.read_csv("C:/Users/rss188/Desktop/Python_CognitiveVulnerabilities/
 ↪Model/Results.csv")
```

8 Plot the results from the baserun

This section focuses on visualizing and saving graphical representations of data related to 'Cognitive Vulnerabilities' and 'Family Support' over a series of years. It begins by creating a dedicated 'Images' folder within a specified directory for storing the generated plots. Utilizing the Matplotlib library, the script sets up a plot with a specific size and plots two datasets: 'Cognitive Vulnerabilities' and 'Family Support', each distinguished by different line styles and widths for clarity. A title, labels for the X-axis, and a legend are added to enhance understanding of the plot. The grid is intentionally removed for a cleaner visual presentation. Finally, the script saves the generated plot as an image file in the created 'Images' folder and displays it. This approach efficiently encapsulates the process of data visualization and storage, providing a straightforward method for analyzing and presenting time-series data.

[8]:
```
# Create 'Images' folder in the specified directory
home_directory = 'C:/Users/rss188/Desktop/Python_CognitiveVulnerabilities'
image_dir = os.path.join(home_directory, 'Images')
os.makedirs(image_dir, exist_ok=True)

# Set the size of the plot
plt.figure(figsize=(15,6))

# Plot: Cognitive Vulnerabilities with thicker lines
plt.plot(np.array(baserun['Years']), np.array(baserun['Cognitive␣
 ↪Vulnerabilities']),
         'black', label='Cognitive Vulnerabilities', linewidth=2.5)

# Plot: Family Support with thicker lines
plt.plot(np.array(baserun['Years']), np.array(baserun['Family Support']),
         'k--', label='Family Support', linewidth=2.5)  # 'k--' represents␣
 ↪dashed black line
```

```
# Common Title, Labels, and Legend
plt.title("Cognitive Vulnerabilities and Family Support (Baserun)")
plt.xlabel("Age")
plt.legend()

# Remove grid
plt.grid(False)

# Save the combined plot
plt.savefig(os.path.join(image_dir, 'Cognitive Vulnerabilities and Family␣
 ↪Support_Baserun.png'))

# Display the plot
plt.show()
```



9 Run the model for multiple case scenarios

Here, a Python script automates the process of running multiple case simulations using Stella software. Initially, the script reads parameters for each case from an Excel file into a DataFrame. For each case, it then writes these parameters into a 'Parms.csv' file, and subsequently runs the Stella model simulation through a subprocess call, using a specified path to the Stella Architect executable. The results of each simulation are read from a generated CSV file and appended to a master DataFrame, all_results, with a unique identifier for each case. After processing all cases, the consolidated results are saved to 'All_Cases_Results.csv'. The script efficiently handles multiple simulations and collates their outputs, demonstrating an effective approach for batch processing in system dynamics models.

[9]:
```
# Read the cases from an Excel file into a DataFrame
df = pd.read_excel('C:/Users/rss188/Desktop/Python_CognitiveVulnerabilities/
 ↪Model/Cases.xlsx')

# Define the path to the Stella simulator executable
stella_simulator_path = "C:/Program Files/isee systems/Stella Architect/Stella␣
 ↪Architect.exe"

# Initialize an empty DataFrame to store results from all cases
all_results = pd.DataFrame()

# Loop through each row (case) in the DataFrame
for index, row in df.iterrows():

    # Open the 'Parms.csv' file and write parameters for the current case
    with open('C:/Users/rss188/Desktop/Python_CognitiveVulnerabilities/Model/
 ↪Parms.csv', 'w') as file:
        for col in row.index:
            # Write each parameter and its value in the format 'parameter:␣
 ↪value'
            file.write(f"{col}, {row[col]}\n")

    # Execute the Stella model simulation for the current case using subprocess
    subprocess.run([stella_simulator_path, "-r",
                    ("C:/Users/rss188/Desktop/Python_CognitiveVulnerabilities/
 ↪Model/"
                     "Developmental-Transitions-2-2-10.stmx")])

    # Read the results of the simulation from a CSV file into a DataFrame
    baserun_results = pd.read_csv("C:/Users/rss188/Desktop/
 ↪Python_CognitiveVulnerabilities/Model/Results.csv")

    # Add a new column 'Case' to baserun_results to label the results with the␣
 ↪case number
    baserun_results['Case'] = f'{index+1}'

    # Append the results of the current case to the all_results DataFrame
    all_results = pd.concat([all_results, baserun_results], ignore_index=True)

    # Print a message indicating completion of processing for the current case
    print(f"Case {index+1} completed.")

# Save the combined results from all cases to a CSV file
all_results.to_csv('All_Cases_Results.csv', index=False)
```

```
# Print a final message indicating all cases have been processed and results␣
 ↪are saved
print("All cases have been processed and results saved.")
```

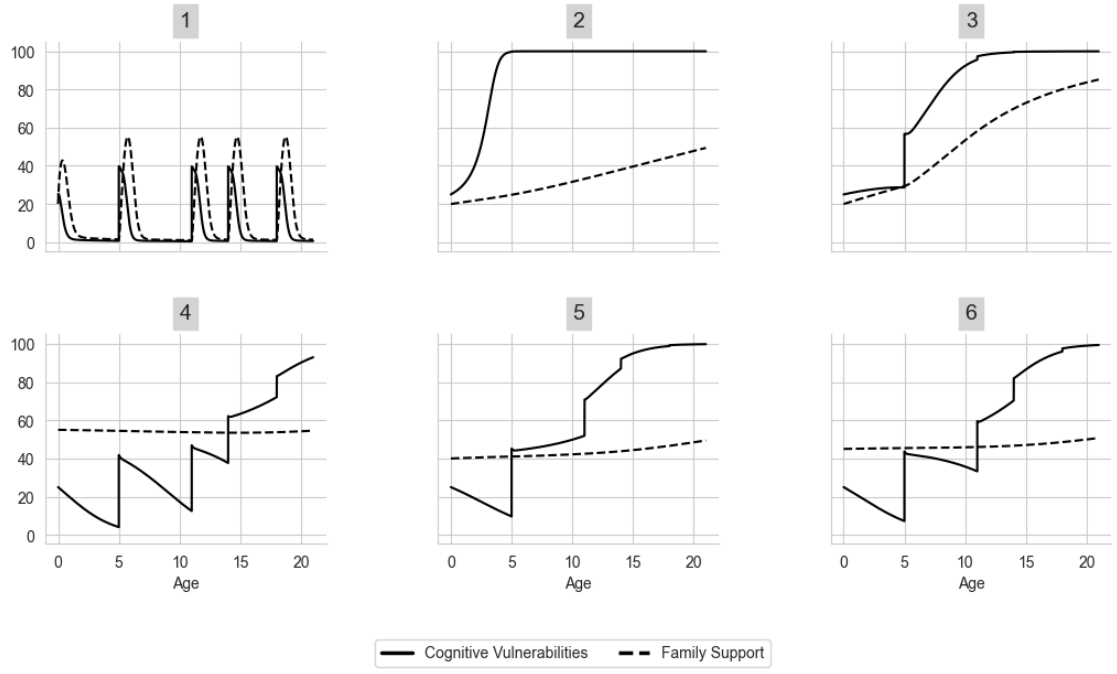
```
Case 1 completed.
Case 2 completed.
Case 3 completed.
Case 4 completed.
Case 5 completed.
Case 6 completed.
All cases have been processed and results saved.
```

10 Plot the results from the case scenarios

This segment uses Matplotlib and Seaborn to create facet plots for visualizing 'Cognitive Vulnerabilities' and 'Family Support' data across various cases. It sets a white grid aesthetic style, loads data from a CSV file, and generates a FacetGrid for organized plotting. 'Cognitive Vulnerabilities' and 'Family Support' are plotted with distinct styles, while the X-axis is labeled as 'Age' for context. Y-axis titles are removed to declutter the plot. Custom titles enhance readability, and a legend explains the data. The resulting plot is saved in an 'Images' folder, providing a precise, organized visualization of multi-case data for analysis.

[10]:
```
# Set the aesthetic style of the plots
sns.set_style("whitegrid")

# Read the combined results
all_results = pd.read_csv('All_Cases_Results.csv')

# Create a FacetGrid and increase the size of each facet
facet_size = 3.5  # Adjust this value as needed
g = sns.FacetGrid(all_results, col='Case', col_wrap=3, sharex=True,␣
 ↪sharey=True, height=facet_size)

# Plot 'Cognitive Vulnerabilities' with solid black line
g.map(sns.lineplot, 'Years', 'Cognitive Vulnerabilities', color='black', lw=1.5)

# Plot 'Family Support' with dashed black line
g.map(sns.lineplot, 'Years', 'Family Support', color='black', lw=1.5,␣
 ↪linestyle='--')

# Update titles and keep y-axis tick labels, but do not set a y-axis title
for ax, title in zip(g.axes.flatten(), g.col_names):
    ax.set_ylabel('')  # Not setting a specific y-axis title
    ax.set_xlabel('Age')  # Set X-axis label to 'Age'
    new_title = str(title).replace('Case_', 'Case ')  # Convert title to string␣
 ↪and change title format
```

```
    ax.set_title(new_title, fontsize=14, backgroundcolor='lightgrey', pad=10) ␣
 ↪# Customize title

# Adjust layout to make room for legend at the bottom
plt.subplots_adjust(top=0.9, bottom=0.25, hspace=0.4, wspace=0.4)

# Create a custom legend and place it at the bottom of the facet wrap
legend_labels = ['Cognitive Vulnerabilities', 'Family Support']
legend_lines = [plt.Line2D([0], [0], color='black', lw=2.5),
                plt.Line2D([0], [0], color='black', lw=2.5, linestyle='--')]
g.fig.legend(handles=legend_lines, labels=legend_labels, title='',
             loc='lower center', bbox_to_anchor=(0.52, 0.08), ncol=2)

# Define the image directory
image_dir = 'C:/Users/rss188/Desktop/Python_CognitiveVulnerabilities/Images'

# Check if directory exists and create if not
os.makedirs(image_dir, exist_ok=True)

# Save the combined plot
plt.savefig(os.path.join(image_dir, 'Cognitive Vulnerabilities and Family␣
 ↪Support Across Cases.png'))

# Display the plot
plt.show()
```

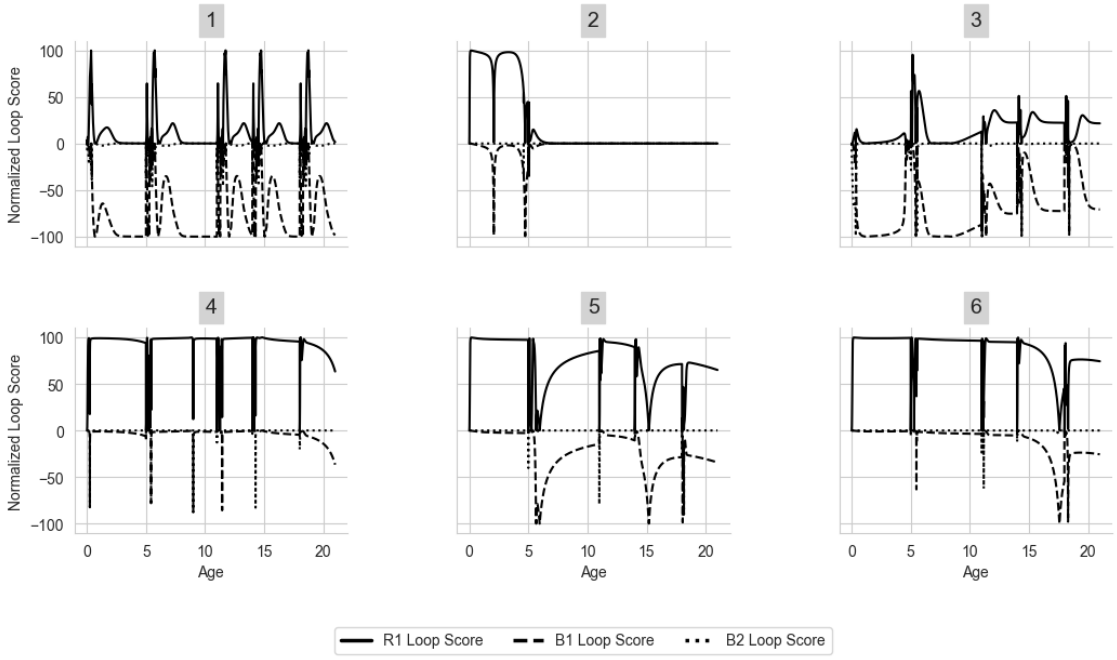


11 Plot the loop scores of the three major loops

In this code, the aesthetic style for the plots is set, and data is imported from a CSV file into a DataFrame. A grid of subplots is created, with each one representing an unique 'Case,' and data is visualized using different line styles, including solid, dashed, and dotted lines. The subplots are customized with titles, X-axis labels ('Age'), and Y-axis labels ('Normalized Loop Score') for improved clarity. Layout adjustments are implemented to accommodate a legend at the bottom, and a custom legend is generated to differentiate the various line styles utilized in the subplots. Lastly, the combined plot is saved as an image file and displayed, providing a clear representation of normalized loop scores across different cases.

[11]:
```
# Set the aesthetic style of the plots
sns.set_style("whitegrid")

# Read the combined results
all_results = pd.read_csv('All_Cases_Results.csv')

# Create a FacetGrid and increase the size of each facet
facet_size = 3.5  # Adjust this value as needed
g = sns.FacetGrid(all_results, col='Case', col_wrap=3, sharex=True,␣
 ↪sharey=True, height=facet_size)

# Plot 'Cognitive Vulnerabilities' with solid black line
g.map(sns.lineplot, 'Years', 'R1 Loop Score', color='black', lw=1.5)

# Plot 'Family Support' with dashed black line
g.map(sns.lineplot, 'Years', 'B1 Loop Score', color='black', lw=1.5,␣
 ↪linestyle='--')

# Plot 'Family Support' with dashed black line
g.map(sns.lineplot, 'Years', 'B2 Loop Score', color='black', lw=1.5,␣
 ↪linestyle=':')

# Update titles and keep y-axis tick labels, but do not set a y-axis title
for ax, title in zip(g.axes.flatten(), g.col_names):
    ax.set_ylabel('Normalized Loop Score')  # Set Y-axis title
    ax.set_xlabel('Age')  # Set X-axis label to 'Age'
    new_title = str(title).replace('Case_', 'Case ')  # Convert title to string␣
 ↪and change title format
    ax.set_title(new_title, fontsize=14, backgroundcolor='lightgrey', pad=10) ␣
 ↪# Customize title

# Adjust layout to make room for legend at the bottom
plt.subplots_adjust(top=0.9, bottom=0.25, hspace=0.4, wspace=0.4)

# Create a custom legend and place it at the bottom of the facet wrap
legend_labels = ['R1 Loop Score', 'B1 Loop Score','B2 Loop Score']
```

```
legend_lines = [plt.Line2D([0], [0], color='black', lw=2.5),
                plt.Line2D([0], [0], color='black', lw=2.5, linestyle='--'),
                plt.Line2D([0], [0], color='black', lw=2.5, linestyle=':')]
g.fig.legend(handles=legend_lines, labels=legend_labels, title='',
             loc='lower center', bbox_to_anchor=(0.52, 0.08), ncol=3)

# Define the image directory
image_dir = 'C:/Users/rss188/Desktop/Python_CognitiveVulnerabilities/Images'

# Check if directory exists and create if not
os.makedirs(image_dir, exist_ok=True)

# Save the combined plot
plt.savefig(os.path.join(image_dir, 'Loop Scores Across Cases.png'))

# Display the plot
plt.show()
```



12 Record the end time

This code records the end time after the script finishes, and determines the amount of time it takes to run the entire code by calculating the difference between the end time and start time. This calculation is saved as a CSV file in the home directory and printed in the Jupyter Notebook.

[12]:
```
end_time = time.time()
execution_time = end_time - start_time

# Define path to the home directory
home_directory = 'C:/Users/rss188/Desktop/Python_WorkAttrition'

# Create the file path for the CSV file in the home directory
csv_file_path = os.path.join(home_directory, 'Execution_time.csv')

# Writing the execution time to a CSV file in the home directory
with open(csv_file_path, 'w') as file:
    file.write('Execution Time (seconds)\n')
    file.write(f'{execution_time}\n')

print(f"Execution time: {execution_time} seconds")
```

```
Execution time: 74.49540662765503 seconds
```