Capstone 4G06: System Verification and Validation Plan for yoGERT GIS Toolbox

Team 19,
Smita Singh, Abeer Alyasiri, Niyatha Rangarajan,
Moksha Srinivasan, Nicholas Lobo, Longwei Ye

New VnV Template

November 2, 2022

1 Revision History

Date	Version	Notes
November 2, 2022	1.0	Longwei: Sec 3; Moksha: Sec 6,7; Smita: Sec 6,7; Abeer: Sec 3,4,formatting; Niyatha: 5.2; Nicholas: Sec 5.2,5.3
March 6, 2023	2.0	Abeer: Unit Test (6.2.1-6.2.5) Moksha: Unit Test (6.2.9) Longwei Ye: Non-functional Tests (6.3)Nicholas: Unit Test (6.2.5) Smita Singh: Unit Tests (6.7, 6.10, 6.11)

Contents

List of Tables

2 Symbols, Abbreviations and Acronyms

symbol	**description**
SRS	Software Requirements Specification
GPS	Global Positioning Systems
GIS	Geographical Information Systems
GERT	GIS-based episode reconstruction toolkit
Point	location coordinate with time stamp.
Session	Object activity history quantified by GPS points
Episode	Session
Segment	Group of GPS Points combined based on episode attributes.
Trip	GPS points represents an object moving to a different position.
Route	Object path to get from position A to position B
Mode Detection (MD)	Detection of type of transportation being used
Time Use Diary (TUD)	Time Use Diary are records of continuous events and actions through a particular period of time (usually 24 to 48 hours)
Route Choice Analysis (RCA)	Analyzes route selection from point a to point b
.shp	.shp are geospatial data format files
CSV/.csv	Comma Separated Values is a file type that contains large amounts of data separated by commas.
Potential Activity Locations (PALS)	PALS are potential trip stops
Activity Locations (ALs)	ALs are trip stops
FR	Functional Requirement
NFR	Non-Functional Requirement
DD	Decimal degrees (DD) is a notation for expressing latitude and longitude geographic coordinates as decimal fractions of a degree
DMS	DMS is a notation for expressing altitude and longitude as degrees (°), minutes ('), seconds (")

This document provides the project's verification and validation plan for documentation phase and implementation phase. Also, it will include detailed testing decisions for system test cases and unit test cases.

3 General Information

3.1 Summary

The software being verified and validated is the yoGERT toolbox. The software general functions include:

- Process user's GPS files into compatible data types.
- Determine choice model estimations.
- Extract travel episodes variables.
- Extract segments from travel episodes and classify segments.
- Categorize movement and stop behaviour.

3.2 Objectives

The objectives of the verification and validation plan ordered by importance are:

1. To demonstrate the fundamental functions meet the stakeholders goals.
2. To build confidence in toolbox accessibility and transferability.
3. To build confidence in the toolbox correctness. By confirming the functions are executed as expected by the requirements.
4. To demonstrate the project scope meets the capstone deadline.

   Demonstrating that the fundamental functions meet the stakeholders' goals is the most important objective because it helps to ensure that the project being developed aligns with the expectations of the stakeholders. This will help improve the overall quality of the products as it will be designed with the end-users in mind. To demonstrate the project scope meets the capstone deadline is the least important objective as it will be completed as long as the other objectives above it are met.

3.3 Relevant Documentation

The test plan are created follow the documents listed below:

- Software Requirements Specification.
- Module Guide.
- Module Interface Specification.

4 Plan

This section outlines verification and validation plan including details on possible testing approaches and division of resources. The section provides rationalized decisions making process for the verification and validation plan.

4.1 Verification and Validation Team

The testing team consists of all members of yoGERT team. All team members need to be actively aware of each testing plan and involved in all aspects of the testing process. All team members are involved because this is a capstone project that requires students to participate in all the project's stages. Therefore, the team aims to evenly split testing and preparing for automated testing. The table below shows the division of responsibilities. ~~It assigns leaders for different testing milestones.~~ This way the team is sure that every testing stage is on track.

Team Member	**Testing Role**	**Responsibilities**
Abeer Alyasiri	SRS and Design Verification and ~~Acceptance~~ Unit Testing	Leads document and code walkthroughs and inspections. Leads user testing and ~~business acceptance~~ SRS verification using black box techniques and white box techniques for automated tests.
Longwei Ye	Integration Testing, Stress Testing, Regression Testing, and Performance Testing	Leads regression, integration, and stress testing using black box techniques.
Moksha Srinivasan	Unit Testing , system testing, and user testing	Leads implementation of white box testing techniques for automated testing. Black box technique for manual testing.
Smita Singh	Unit Testing	Leads implementation of white box testing techniques.
Nicholas Lobo	~~System~~ Unit Testing	~~Leads functional requirements testing.~~ Leads implementation of white box testing techniques.
Niyatha Rangarajan	~~System~~ Unit Testing	~~Leads non-functional requirements testing.~~ Leads implementation of white box testing techniques.

Table 1: Verification and Validation Team.

4.2 SRS Verification Plan

The SRS verification plan consists of three parts. The main objectives is to check if the SRS document was completed according to the Volere Template Standards and if the requirements address the project goals. All parts will

utilize a static testing technique that includes structured and unstructured reviews, walkthroughs or checklists.
Part one is the unstructured feedback received from classmates in the form of GitHub issues. These reviews provide technical improvements on the document from outside the team. It is helpful because it improves the document's information flow to professionals in the field, such as software engineers.
Part two is the structured review received from the TA. The TA follows a checklist in the form of a rubric. The feedback is beneficial because the SRS is reviewed from an industry standard perspective. Hence, the document will be closer to implementing best-practices techniques throughout the document. This increases the productivity of using the SRS during the design stage.
Part three is a structured review with the supervisor. The review will be a combination of SRS walkthrough and modified task based inspection. The objectives of the walkthrough is to introduce the supervisor to the team's documentation and receive general feedback on the scope and clarity of the documentation. On the other hand the task based inspection will analyze both functional and non-functional requirements in depth. The inspection will consist of questions to motivate the supervisor to think about the relationship between system goals and the formulated requirements. This is helpful with removing ambiguities of the requirement's relevance to the desired final system. Also, the inspection will focus on problem categorization. These will represent the talking points of the task based inspection with the supervisor. The categorizations are clarity of requirements, conflicting requirements, and unrealistic requirements problems.
All reviews collected from the SRS verification plan will be applied to the document before the design document deliverable.

4.3 Design Verification Plan

Design verification will be similar to the SRS verification part one and part two. In addition it will include a formal review by teammates using a checklist. The checklist will consist of Dr. Smith's MG and MIS checklists and the following points:

- Each design decision maps to one or more requirements.
- Each design specification has one output.

- Each function decomposition follow top to down design model.
- Design specification connect functional processes logically for the user to carry out tasks.
- Design specification does not include implementation details.
- Design specification describe inputs, logical operations, and output.
- Design specification outputs are consistent across a division of input cases.
- Design specification outlines error responses for unexpected behaviour.

The team will conduct the verification against the checklist using static and dynamic testing techniques. Static testing will involve a walkthrough to proof traceability and accuracy of the system architectural model. Dynamic testing will include unit testing, white box testing, black box testing, and integration testing.

4.4 Verification and Validation Plan Verification Plan

Verification and validation plan will be verified through reviews. It is important to highlight that it is difficult to proof the correctness of the test cases. Therefore, the combination multiple verification techniques induct that the verification and validation plan approximately tested critical points of the system.
First, it will be verified against Dr. Smith's VnV-checklist by yoGERT team members. It will verify the completeness of the test cases. It is done by tracing at least one requirement to a test case and examining the requirement across different types of inputs.
Second, it will be reviewed by classmates in an informal way. This feedback is beneficial because it is outside the team professional opinion on what information is missing from the document.
Third, it will be reviewed by the TA in a standard way using the rubric as a checklist. The review will focus on the accuracy of information used to formulate the plan and if the plan is appropriate to the project. The plan is appropriate if it is feasible within the capstone timeline and test cases are complete within its requirement scope.

4.5 Implementation Verification Plan

The implementation verification plan includes both system test cases and unit test cases listed in this document. The verification plan is a combination of different testing techniques to start with testing the building blocks of the system up to testing structural interaction between theses components.
In the early stages of the implementation verification plan, the team will conduct static verification techniques. It includes code inspections to test code readability and code walkthroughs to verify implementation meets that design specification.
The other stages of the implementation verification plan will rely on dynamic testing techniques. These tests will be driven by white box testing, stress testing, regression testing, and integration testing techniques. These techniques are focused on proving that the system follows the design specification and requirements specification and is consistent with the addition of new components. Also, The verification plan must encapsulate testing scenarios of how the system reacts to faulty inputs. It can be tested by inputting irrational data points and observing if a safe output will be produced instead of system failure. Therefore, it is important that the test cases will include boundary and edge inputs to the system's safe outputs and consistent behaviour.

4.6 Automated Testing and Verification Tools

The section was done in the development plan document Sections 6 and 7.

The details of this section will likely evolve further in the project. Currently the plan is to only use automation testing for unit tests.

4.7 Software Validation Plan

Software validation plan will be divided into two parts. Part one will involve walkthrough and task based inspection with the supervisor similar to the SRS verification plan section. It will be conducted, for the same reasons from before, to flush out any problems with the SRS requirements. This standardized review will be conducted prior to implementation. Part two will involve walkthrough and demonstration to the supervisor to validate that the system behaves as the primary stakeholder expected. The formal walkthrough ensures validation of the design implementation functionality.

On the other hand the demonstration validate the system's usability and response to user inputs. The supervisor will be able to provide feedback as he understand the GIS toolbox functionality and he represents a typical user for the yoGERT toolbox. If time permits, external data can be used for validation. The external data will be ARC GIS outputs to the same inputs fed into the yoGERT toolbox. The objective of this validation is to show the consistency between the yoGERT toolbox and the current available toolbox. This form of validation need to use external data with exact method applied to since the yoGERT toolbox is implementing parts of the ARC GIS application. Hence, not all outputs of the ARC GIS application are the expected outputs from the yoGERT toolbox.

5 System Test Description

5.1 Tests for Functional Requirements

The testing of functional requirements will be divided into two sections. One to test user functionality and one to test system functionality.

5.1.1 User Functionality Tests

This type of software testing that focuses on verifying the functionality of a system or application from the end-user's perspective. The purpose of this testing is to ensure that the software meets the needs and expectations of its intended users by testing all user-facing features. How data is being read will be the main focus of these tests to ensure a high level of user satisfaction with the final product.

User Input Testing

1. test-UT-1

   Control: Manual

   Initial State: The application has been loaded onto the computer

   Input: User loads in a CSV file of GPS data

   Output: The system saves the CSV file of GPS data

Test Case Derivation: The user wants the application to read the given CSV file

How test will be performed: Different sets of valid CSV data will be uploaded by the tester to see if the computer reads the values correctly

Associated Functional Requirement: FR1

2. test-UT-2

   Control: Manual

   Initial State: The application has been loaded onto the computer

   Input: The system has a loaded file of GPS data

   Output: The system saves the values found in the CSV file as latitude longitude and time variables

   Test Case Derivation: The user wants to use the software to save the given CSV files into variables that can be manipulated

   How test will be performed: Different sets of valid CSV's of GPS data will be uploaded by the tester to see if the computer reads the values correctly

   Associated Functional Requirement: FR2,FR5

3. test-UT-3

   Control: Manual

   Initial State: The application has been loaded onto the computer

   Input: User loads in a CSV file of time use diaries (TUD)

   Output: The system saves the the the CSV file of TUD's

   Test Case Derivation: The user wants to use the software to read the given CSV file and save it

   How test will be performed: Different sets of valid CSV's of TUD data will be uploaded by the tester to see if the computer reads the values correctly

   Associated Functional Requirement: FR3

4. test-UT-4

   Control: Manual

   Initial State: A CSV of GPS data has been inputted to the application

   Input: User downloads the file that it uploaded to the system

   Output: The system gives the user the data in a CSV format

   Test Case Derivation: The user wants to use the software to read the given CSV file and save it to attributes that can be

   How test will be performed: Different sets of valid CSV's of TUD data will be uploaded by the tester to see if the computer reads the values correctly

   Associated Functional Requirement: FR4

5.1.2 System Functionality Tests

These tests will have a focus on verifying the internal workings of the system. It will involve testing the individual components and subsystems that make up the system to ensure that they function as intended and interact with each other properly. How data is being processed and outputted will be the main focus of these tests with the goal of identifying and fixing any defects or issues that could affect the overall performance and stability of the system.

System Output Testing

1. test-ST-1

   Control: Manual

   Initial State: CSV of GPS data has been inputted to the application

   Input: The user types a function to call for the system to organize the inputted data into episodes

   Output: The system returns a report of categorized data points such as speed, duration, distance, and change in direction.

   Test Case Derivation: The system needs to be displayed in a way that the user can read easily

How test will be performed: The tester will use a variety of CSV files filled with valid GPS data and use the function call to see if valid reports were generated

Associated Functional Requirement: FR6

2. test-ST-2

   Control: Manual

   Initial State: A CSV of TUD data has been inputted to the application

   Input: The user types a function to call for the system to organize the inputted data into episodes

   Output: The system returns a report which contains a list of episodes that have categorized data points such as speed, duration, distance, and change in direction.

   Test Case Derivation: The system needs valid GPS points

   How test will be performed: The tester will use a variety of CSV files filled with valid GPS data and use the function call to see if valid reports were generated

   Associated Functional Requirement: FR7

3. test-ST-3

   Control: Manual

   Initial State: CSV of GPS data has been inputted to the application

   Input: The user types a function to call for the system to organize the inputted data into episodes

   Output: The system returns a report of episodes categorized by different methods of transportation(walk, car, bus).

   Test Case Derivation: The user wants to understand the methods of travel used from the set of data points given

   How test will be performed: The tester will use a variety of CSV files filled with valid GPS data and use the function call to see if valid categories are found in the reports generated

Associated Functional Requirement: FR8,FR20

4. test-ST-4

   Control: Manual

   Initial State: CSV of GPS data has been inputted to the application and a report of episodes was generated

   Input: The user selects one of the episodes generated from the report

   Output: The system returns the segments of the episodes into type stop and trip

   Test Case Derivation: The user wants to understand the behaviour of the object given an episode in the report

   How test will be performed: The tester will use a variety of generated reports to see if valid episode segments were created

   Associated Functional Requirement: FR9,FR19

5. test-ST-5

   Control: Manual

   Initial State: CSV of GPS data has been inputted to the application

   Input: The report of episodes and segments are generated

   Output: The system generates the trip trajectory values based on the given segments

   Test Case Derivation: The system needs trip trajectory values for route choice analysis

   How test will be performed: The tester will validate the trajectory values based on the given CSV GPS data

   Associated Functional Requirement: FR10

6. test-ST-6

   Control: Manual

   Initial State: CSV of GPS data has been inputted to the application and the report is generated

Input: The report of episodes and segments are generated

Output: The system generates and stores activity locations for each of the episodes in the report

Test Case Derivation: The system needs to generate high and low activity locations

How test will be performed: The tester will validate a sample reports activity location matches with a curated list of episodes with known activity locations

Associated Functional Requirement: FR11,FR16,18

7. test-ST-7

   Control: Manual

   Initial State: CSV of GPS data has been inputted to the application and the report of episodes and their segments are generated

   Input: The trajectory values are calculated by the system

   Output: The system creates RCA variables based on the trip trajectory

   Test Case Derivation: The system needs the RCA variables to define route choice behaviour data set.

   How test will be performed: The tester will generate multiple RCA datasets from different reports and check the validity of them

   Associated Functional Requirement: FR12,FR13

8. test-ST-8

   Control: Manual

   Initial State: A RCA dataset has been generated by the software

   Input: The user request a route from two GPS points A and B

   Output: The system generates a mapped route from position A and position B

   Test Case Derivation: The user needs requested routes given two GPS points

How test will be performed: The tester will request for multiple routes to be created from a generated RCA dataset

Associated Functional Requirement: FR14,FR17

9. test-ST-9

   Control: Manual

   Initial State: A RCA dataset has been generated by the software

   Input: The user request a route from two GPS points A and B with selected constraints

   Output: The system generates a mapped route from position A and position B with selected constraints

   Test Case Derivation: The user wants customized routes based on selected constraints

   How test will be performed: The tester will request for multiple routes with selected constraints be created from a generated RCA dataset

   Associated Functional Requirement: R15

10. test-ST-10

    Control: Manual

    Initial State: Overpy API Server unavailable

    Input: Valid input and output file paths and valid input CSV file

    Output: Log warning displaying for which stop points the server was not able to fetch activity locations

    Test Case Derivation: The user wants to be informed if there are stop points for which activity locations are not fetched

    How test will be performed: The tester will disconnect from the internet, and run fetch Activity Locations module with valid inputs and checks that all the stop points in log warnings do not show up in the generated output CSV file

    Associated Functional Requirement: R8, R14

5.2 Tests for Nonfunctional Requirements

5.2.1 Appearance Tests

The information must be presented to the user in readable format. Hence, there will be appearance related tests.

1. test-id1

   Type: Static/Manual Testing

   Initial State: User messages are present and users can easily see what errors occurred in a clear and uncomplicated way.

   Input/Condition: The user applies a function to some incorrect data.

   Output/Result: The user can easily decipher the error message provided

   How test will be performed: The user will be provided incorrect input and told to explain what went wrong and how they could fix it in the future. If successful, error messages are easily decipherable.

   Associated NFR: 6, 7, 21

2. test-id2

   Type: Manual testing

   Initial State: CSV of GPS data has been inputted to the application

   Input: The user types a function to call for the system to organize the inputted data into episodes

   Output: The system returns a possible set of input data types if the function matches a stored function in the system

   How test will be performed: This works like VS code, were as you type a function, a function description hovers over the function call, depciting the required user input for that function. One can try this test with different function calls to check its validity.

   Associated NFR: 1, 2, 3, 4, 5, 6, 7

   Associated NFR: 1, 2, 3, 4, 5, 6, 7

5.2.2 Large Data Memory and Performance

The toolbox must work with large sets of data, hence test must consider the edge cases of data size and its relevant processing time.

1. test-id4

   Type: Regression testing

   Initial State: CSV of 47.3 million data points of GPS data has been inputted to the application

   Input/Condition: The user types a function to call for the system to organize the inputted data (word document) into episodes giving a word document as input.

   Output/Result: The system returns a report of categorized data points such as speed, duration, distance, and change in direction within 6000 seconds upon request

   How test will be performed: We perform edge case tests to see if performance and capacity requirements are met.

   Associated NFR: 9, 11, 12

2. test-id5

   Type: Unit testing

   Initial State: CSV of GPS data has been inputted to the application

   Input: The user types a function to call for the system to organize the inputted data into episodes

   Output: The system returns a possible set of input data types if the function matches a stored function in the system

   How test will be performed: We perform a unit test to see if the outputted data matches the expected episodes we require from the system. This is helpful for precision requirements.

   Associated NFR: 11

5.2.3 User information Security and Reliability

Since the information inputted will be used by APIs online, the system must ensure protection of user information.

1. test-id6

   Type: Manual

   Initial State: CSV of GPS data has been inputted to the application

   Input/Condition: The user types a function to call for the system to organize the inputted data into episodes

   Output/Result: No data seen at API endpoint.

   How test will be performed: We must make sure we use online APIs like pandas, geopy, etc. does not store any user inputted information.

   Associated NFR: 19

2. test-id7

   Type: Dynamic testing

   Initial State: CSV of GPS data has been inputted to the application

   Input: The user types a function to call for the system to organize the inputted data into episodes

   Output: Inputted data has not changed once the episodes are created.

   How test will be performed: Black box testing using Finite state machines. If there is no change of state for the input, then the test succeeds.

   Associated NFR: 10

5.2.4 Environment issues

For the functioning of the application, it must have certain prerequisite software like Python installed and the environment it is run on like Mac, Windows, etc. must be accounted for.

1. test-id8

   Type: Unit testing

   Initial State: CSV of GPS data has been inputted to the application

   Input/Condition: The user types a function to call for the system to organize the inputted data into episodes.

   Output/Result: Error is outputted stating that Python must be installed in the system.

   How test will be performed: Python is not installed in the system before inputting the data.

   Associated NFR: 15

2. test-id9

   Type: Unit testing

   Initial State: CSV of GPS data has been inputted to the application

   Input/Condition: The user types a function to call for the system to organize the inputted data into episodes.

   Output/Result: The system returns a report of categorized data points such as speed, duration, distance, and change in direction

   How test will be performed: Python2 is installed in the system before inputting the data. Since, Python can only be installed on a valid OS, we simultaneously test for the operational environment.

   Associated NFR: 14,15,16

5.2.5 Accessibility issues

3. test-id10

   Type: Manual testing

   Initial State: CSV of GPS data has been inputted to the application

   Input/Condition: The user typed a function to call for the system to output the inputted data into episodes with input to specify file output.

Output/Result: The system returns a report of categorized data points such as speed, duration, distance, and change in direction in a csv file format to allow output to be saved

How test will be performed: The GPS data points are inputs and a function is ran on them with the option to open the output in csv file format.

Associated NFR: 8.

5.2.6 Scalability issues

4. test-id11

   Type: Manual testing

   Initial State: application has a pre-existing data points as inputs

   Input/Condition: The second user inputs a different set of GPS data points.

   Output/Result: The system returns a successful message of the accepted inputs and maps them as a continuation of the initial data points.

   How test will be performed: The toolbox initialized with an input then an new input is loaded on the toolbox.

   Associated NFR: 13.

5.2.7 Security issues

NFRs 17,18 are not applicable to the yoGERT toolbox as it was mentioned in the Hazard Analysis document. After the Hazard Analysis NFR changes are applied to the SRS document then appropriate test cases will be described here.

5.3 Traceability Between Test Cases and Requirements

	FR1	FR2	FR3	FR4	FR5	FR6	FR7	FR8	FR9	FR10	FR11	FR12	FR13	FR14	FR15	FR16	FR17	FR18	FR19	FR20
test-UT-1	X				X															
test-UT-2		X																		
test-UT-3			X																	
test-UT-4				X																
test-ST-1						X														
test-ST-2							X													
test-ST-3								X												X
test-ST-4									X										X	
test-ST-5										X										
test-ST-6											X					X		X		
test-ST-7												X	X							
test-ST-8														X			X			
test-ST-9															X					

Table 2: Traceability Matrix Showing the Connections Between Functional Requirements and their test.

	NFR1	NFR2	NFR3	NFR4	NFR5	NFR6	NFR7	NFR8	NFR9	NFR10	NFR11	NFR12	NFR13	NFR14	NFR15	NFR16	NFR17	NFR18	NFR19
test-id1											X								
test-id2	X	X	X	X	X	X	X												
test-id3									X		X	X							
test-id4											X								
test-id5																			X
test-id6										X									
test-id7															X				
test-id8														X	X	X			
test-id9								X											
test-id10													X						

Table 3: Traceability Matrix Showing the Connections Between Non Functional Requirements and their test.

6 Unit Test Description

The pytest library will be used to complete unit testing for this toolbox. To develop unit tests for the internal functions of the program, we will be creating a corresponding test file for each module. Each test file will contain unit tests for each function within the module. These tests contain a variety of inputs, including those which output the correct transformation as well as inputs that generate errors and exceptions.

We will be using coverage metrics to determine how well-tested our code is. This will be determined through the use of coverage.py, a python library that quickly analyzes code coverage of all modules within a project. We will be aiming for 90% code coverage per module, ensuring that we adequately test all functions.

6.1 Unit Testing Scope

Route choice analysis variable modules will be verified for correct functionality (correct sample inputs output correct sample outputs), but logic of previously existing modules will be assessed for correctness by our supervisor, Dr. Paez.

6.2 Tests for Functional Requirements

~~This section will be completed once the MIS has been updated and there is greater clarity on specific modules.~~

6.2.1 Network Graph Module Test Cases

Test 6.2.1.1:	Creating Network Graph from GPS pings of a trace.
Requirements:	R11,R4
Description:	Tests if the network graph functionality works when given a trace with GPS pings.
Type:	Unit test (static, automated)
Initial State:	The user has ran preprocessing on their initial GPS data set and episode generation on the output from the preprocessing module.
Input:	File path to the preprocessing trace CSV output and mode of transportation.
Output:	NetworkGraph object.
Pass:	NetworkGraph object contains an attribute for a successfully created networkx.MultiDiGraph object.

Test 6.2.1.2:	Creating Network Graph from GPS pings of episode.
Requirements:	R11,R4
Description:	Tests if the network graph functionality works when given a episode with GPS pings.
Type:	Unit test (static, automated)
Initial State:	The user has ran preprocessing on their initial GPS data set and episode generation on the output from the preprocessing module.
Input:	File path to the episode CSV output and mode of transportation.
Output:	NetworkGraph object.
Pass:	NetworkGraph object contains an attribute for a successfully created networkx.MultiDiGraph object.

Test 6.2.1.3:	Getting nearest graph node of a GPS coordinate
Requirements:	R11,R4
Description:	Tests if the network graph get function works in getting the nearest node when the coordinate is included in the graph's area.
Type:	Unit test (static, automated)
Initial State:	The user has ran preprocessing on their initial GPS data set, episode generation on the output from the preprocessing module, and created a network graph object using a trace or episode csv file.
Input:	Network Graph object and GPS coordinate as a tuple of (latitude, longitude).
Output:	Node number in integers.
Pass:	Node number is within the Network Graph networkx.MultiDiGraph object.

Test 6.2.1.4:	Getting mode of Network Graph object
Requirements:	R11,R4
Description:	Tests if the network graph get function works in getting the graph's mode of transportation.
Type:	Unit test (static, automated)
Initial State:	The user has ran preprocessing on their initial GPS data set, episode generation on the output from the preprocessing module, and created a network graph object using a trace or episode csv file.
Input:	Network Graph object.
Output:	string for the network transportation mode.
Pass:	Output is the same input entered when the Network Graph object was created.

Test 6.2.1.5:	Finds nearest node of GPS coordinate outside the Network Graph.
Requirements:	R11,R4
Description:	Tests if the functiona catches the out of bound exception.
Type:	Unit test (static, automated)
Initial State:	The user has ran preprocessing on their initial GPS data set, episode generation on the output from the preprocessing module, and created a network graph object using a trace or episode csv file.
Input:	Network Graph object and GPS coordinate as a tuple of (latitude, longitude).
Output:	Exception.
Pass:	Output is an exception of type OutOfBoundsCoordException.

Test 6.2.1.6:	Creating Network Graph from GPS pings with incorrect transportation mode
Requirements:	R11,R4
Description:	Tests if the function catches the exception for the incorrect transportation mode.
Type:	Unit test (static, automated)
Initial State:	The user has ran preprocessing on their initial GPS data set and episode generation on the output from the preprocessing module.
Input:	File path to the preprocessing trace CSV output and mode of transportation.
Output:	Exception.
Pass:	Output is an exception of type InvalidModeException.

Test 6.2.1.7:	Creating Network Graph from GPS pings with empty file path.
Requirements:	R11,R4
Description:	Tests if the function catches the exception for the empty file path.
Type:	Unit test (static, automated)
Initial State:	The user has ran preprocessing on their initial GPS data set and episode generation on the output from the preprocessing module.
Input:	File path to the preprocessing trace CSV output and mode of transportation.
Output:	Exception.
Pass:	Output is an exception of type EmptyFilePathException.

6.2.2 Episode Shortest Route Module Test Cases

Test 6.2.2.1:	Creating Episode Shortest Route from Network Graph, algorithm optimizer type, and GPS pings
Requirements:	R11,R4
Description:	Tests if the episode shortest route functionality to find routes when given a network graph, optimizer, and GPS pings of an episode.
Type:	Unit test (static, automated)
Initial State:	The user has ran preprocessing on their initial GPS data set, episode generation on the output from the preprocessing module, and created a network graph object for a generated episode output.
Input:	NetwrokGraph object, string for optimizer type, string for file path to the episode GPS ping CSV file.
Output:	ShortestRouteEpisode object.
Pass:	ShortestRouteEpisode object contains an attribute for list of subroutes where its length less than or equals the number of GPS coordinates - 1 used to created ShortestRouteEpisode object.

Test 6.2.2.2:	Creating Episode Shortest Route from Network Graph, algorithm optimizer type, and GPS pings with sampling customization.
Requirements:	R11,R4
Description:	Tests if the episode shortest route functionality to find routes when given a network graph, optimizer, GPS pings of an episode, and sampling parameters.
Type:	Unit test (static, automated)
Initial State:	The user has ran preprocessing on their initial GPS data set, episode generation on the output from the preprocessing module, and created a network graph object for a generated episode output.
Input:	NetwrokGraph object, string for optimizer type, string for file path to the episode GPS ping CSV file, boolean for sampling, integer for sampling distance.
Output:	ShortestRouteEpisode object.
Pass:	ShortestRouteEpisode object contains an attribute for list of subroutes where its length less than or equals the number of GPS coordinates - 1 used to created ShortestRouteEpisode object.

Test 6.2.2.3:	Creating Episode Shortest Route from Network Graph, algorithm optimizer type, and GPS pings without sampling.
Requirements:	R11,R4
Description:	Tests if the episode shortest route functionality to find routes when given a network graph, optimizer, GPS pings of an episode, and sampling condition.
Type:	Unit test (static, automated)
Initial State:	The user has ran preprocessing on their initial GPS data set, episode generation on the output from the preprocessing module, and created a network graph object for a generated episode output.
Input:	NetwrokGraph object, string for optimizer type, string for file path to the episode GPS ping CSV file, boolean for sampling.
Output:	ShortestRouteEpisode object.
Pass:	ShortestRouteEpisode object contains an attribute for list of subroutes where its length less than or equals the number of GPS coordinates - 1 used to created ShortestRouteEpisode object.

Test 6.2.2.4:	Creating Episode Shortest Route from Network Graph, incorrect algorithm optimizer type, and GPS pings
Requirements:	R11,R4
Description:	Tests if the episode shortest route catches the exception for invalid optimizer type.
Type:	Unit test (static, automated)
Initial State:	The user has ran preprocessing on their initial GPS data set, episode generation on the output from the preprocessing module, and created a network graph object for a generated episode output.
Input:	NetwrokGraph object, string for optimizer type, string for file path to the episode GPS ping CSV file.
Output:	Exception.
Pass:	Output is an exception of type InvalidWeightException.

6.2.3 Trace Shortest Route Module Test Cases

Test 6.2.3.1:	Creating Trace Shortest Route from Network Graph, algorithm optimizer type, and GPS pings.
Requirements:	R11,R4
Description:	Tests if the trace shortest route functionality to find routes works when given a network graph, optimizer, and GPS pings of a trace.
Type:	Unit test (static, automated)
Initial State:	The user has ran preprocessing on their initial GPS data set, episode generation on the output from the preprocessing module, and created a network graph object for a generated episode output.
Input:	NetwrokGraph object, string for optimizer type, string for file path to the trace GPS ping CSV file.
Output:	ShortestRouteTrace object.
Pass:	ShortestRouteTrace object contains an attribute for list of subroutes where its length less than or equals the number of GPS coordinates - 1 used to created ShortestRouteTrace object.

Test 6.2.3.2:	Creating Trace Shortest Route from Network Graph, incorrect algorithm optimizer type, and GPS pings
Requirements:	R11,R4
Description:	Tests if the trace shortest route catches the exception for invalid optimizer type.
Type:	Unit test (static, automated)
Initial State:	The user has ran preprocessing on their initial GPS data set, episode generation on the output from the preprocessing module, and created a network graph object for a generated episode output.
Input:	NetwrokGraph object, string for optimizer type, string for file path to the episode GPS ping CSV file.
Output:	Exception.
Pass:	Output is an exception of type InvalidWeightException.

6.2.4 Alternative Route Module Test Cases

Test 6.2.4.1:	Creating Alternative Route from algorithm optimizer type, and GPS pings
Requirements:	R12,R4
Description:	Tests if the alternative route functionality to find routes when given an optimizer, and GPS pings of an trace.
Type:	Unit test (static, automated)
Initial State:	The user has ran preprocessing on their initial GPS data set.
Input:	string for optimizer type, string for file path to the trace GPS ping CSV file.
Output:	AlternativeRoute object.
Pass:	AlternativeRoute object contains an attribute for list of subroutes where its length equals the number of GPS coordinates - 1 used to created AlternativeRoute object.

Test 6.2.4.2:	Creating Alternative Route from incorrect algorithm optimizer type, and GPS pings
Requirements:	R12,R4
Description:	Tests if the alternative route catches the exception for invalid optimizer type.
Type:	Unit test (static, automated)
Initial State:	The user has ran preprocessing on their initial GPS data set.
Input:	string for optimizer type, string for file path to the trace GPS ping CSV file.
Output:	Exception.
Pass:	Output is an exception of type InvalidWeightException.

6.2.5 Mapping Module Test Cases

Test 6.2.5.1:	Creating a map object for mapping trace route and saving it as a html file.
Requirements:	R3
Description:	Tests if the map object is created successfully as html file.
Type:	Unit test (static, manual)
Initial State:	The user has ran preprocessing on their initial GPS data set, created a network graph object for the preprocessing output, and created a trace shortest route.
Input:	Network Graph object, ShorstestRouteTrace object, string for file path to save the mapping file.
Output:	html file object.
Pass:	html file displays a map with route and point markers as expected from the trace behaviour if mapped on another mapping API.

Test 6.2.5.2:	Creating a map object for mapping episode route and saving it as a html file.
Requirements:	R3
Description:	Tests if the map object is created successfully as html file.
Type:	Unit test (static, manual)
Initial State:	The user has ran preprocessing on their initial GPS data set, ran episode generation on the output from the preprocessing module, created a network graph object for a generated episode output, and created a episode shortest route.
Input:	Network Graph object, ShorstestRouteEpisode object, string for file path to save the mapping file.
Output:	html file object.
Pass:	html file displays a map with route and point markers as expected from the episode behaviour if mapped on another mapping API.

Test 6.2.5.3:	Creating a map object for mapping alternative route and saving it as a html file.
Requirements:	R3
Description:	Tests if the map object is created successfully as html file.
Type:	Unit test (static, manual)
Initial State:	The user has ran preprocessing on their initial GPS data set, created a network graph object for the preprocessing output, and created a alternative route.
Input:	Network Graph object, ShorstestRouteTrace object, string for file path to save the mapping file.
Output:	html file object.
Pass:	html file displays a map with route and point markers as expected from the trace behaviour if mapped on another mapping API.

Test 6.2.5.4:	Creating a map object for mapping activity location markers and saving it as a html file.
Requirements:	R3
Description:	Tests if the map object is created successfully as html file.
Type:	Unit test (static, manual)
Initial State:	The user has ran preprocessing module on their initial GPS data set, ran episode generation module, ran activity location module.
Input:	string for file path to the saved activity location, string for file path to the stop points used for activity location, string for file path to save the mapping file.
Output:	html file object.
Pass:	html file displays a map with activity location markers and point markers as expected from the trace behaviour if mapped on another mapping API.

Test 6.2.5.5:	Creating a map object for mapping markers for episode points and saving it as a html file.
Requirements:	R3
Description:	Tests if the map object is created successfully as html file.
Type:	Unit test (static, manual)
Initial State:	The user has ran preprocessing module on their initial GPS data set, ran episode generation module.
Input:	string for file path to the episode points, string for file path to save the mapping file.
Output:	html file object.
Pass:	html file displays a map with episode point markers as expected from the episode behaviour if mapped on another mapping API.

6.2.6 Episode Generation Module Test Cases

Test 6.2.6.1:	Creating a trace.csv given a trace of gps pings
Requirements:	R5
Description:	Tests if trace.csv is created successfully.
Type:	Unit test (static, automated)
Initial State:	The user has ran preprocessing on their initial GPS data set.
Input:	File path to the processed trace CSV and name of the folder for the trace.
Output:	A folder with the given name created by the user with the trace.csv inside the folder is created.
Pass:	The folder has been created with the trace.csv has been created successfully .

Test 6.2.6.2:	Creating the segments.csv given a trace.csv
Requirements:	R7
Description:	Tests if segments.csv is created successfully.
Type:	Unit test (static, automated)
Initial State:	The user has ran the createTrace method successfully.
Input:	The user runs the createSegment method and gives the path to the trace folder.
Output:	The segment.csv has been created in the trace folder.
Pass:	Segments.csv has been created successfully .

Test 6.2.6.3:	Creating the stops.csv given a segments.csv
Requirements:	R7
Description:	Tests if stops.csv is created successfully.
Type:	Unit test (static, automated)
Initial State:	The user has a segments.csv created using the create-Segments method.
Input:	The user runs the createStops method and gives the path to the trace folder.
Output:	The stops.csv has been created in the trace folder in a sub folder named stops.
Pass:	Stops.csv has been created successfully .

Test 6.2.6.4:	Gets rid of unwanted stops in stops.csv based on stop time
Requirements:	R9
Description:	Tests if stops.csv has removed data that does not meet a certain time criteria given by the user.
Type:	Unit test (static, automated)
Initial State:	The user has a stops.csv created using the createStops method.
Input:	The user runs the cleanStops method and passes the minimum amount of time each stop has to occur.
Output:	The stops.csv has been cleaned to remove any stops that do not meet the time tolerance given by the user.
Pass:	Stops.csv has been dropped all stops that do not have the minimum time tolerance.

Test 6.2.6.5:	Gets rid of unwanted stops in stops.csv based on stop distance
Requirements:	R9
Description:	Tests if stops.csv has removed data that does not meet a certain distance criteria given by the user.
Type:	Unit test (static, automated)
Initial State:	The user has a stops.csv created using the createStops method.
Input:	The user runs the cleanStops method and passes the minimum amount of time each stop has to occur.
Output:	The stops.csv has been cleaned to remove any stops that do not meet the time tolerance given by the user.
Pass:	Stops.csv has been dropped all stops that do not have the minimum time tolerance.

Test 6.2.6.6:	Create episode given a stop.csv and trace.csv
Requirements:	R6
Description:	Tests if episodes are generated based on trace.csv given the stop.csv as parameters.
Type:	Unit test (static, automated)
Initial State:	The user has used the cleanStops method on the stops.csv.
Input:	The user runs the createEpisode method and passes the path to the trace which has the stop.csv.
Output:	A folder that contains episode.csv.
Pass:	The correct amount of episode.csv's are created based on the stop.csv.

Test 6.2.6.7:	Create summary mode given a trace with generated episodes
Requirements:	R15
Description:	Tests if the summary mode csv is created successfully.
Type:	Unit test (static, automated)
Initial State:	The user has created and generated episodes from a given gps trace.
Input:	The user runs the summarymode method and passes the path to the trace folder.
Output:	The summary mode csv file in the trace folder.
Pass:	Summarymode.csv has been created sucessfully.

Test 6.2.6.8:	Create statistics file given a trace with generated episodes
Requirements:	R16
Description:	Tests if the statistic csv is create successfully
Type:	Unit test (static, automated)
Initial State:	he user has created and generated episodes from a given gps trace.
Input:	The user runs the createStats method on the trace folder path.
Output:	Stats.csv has been created in the trace folder.
Pass:	Stats.csv has been created successfully.

6.2.7 Fetch Activity Locations Module

Test 6.2.7.1:	Output file is generated when fetchActivityLocation is called
Requirements:	R14
Description:	Tests specified output file (CSV) is generated when fetchActivityLocation is called
Type:	Unit test (static, automated)
Initial State:	The user has run episode generation
Input:	a path to input CSV containing stop point information and a path to output CSV file
Output:	Output CSV file containing each stop point and corresponding list of Activity locations
Pass:	A specified output CSV file is generated at path given by user

Test 6.2.7.2:	Input file invalid error generated
Requirements:	R8
Description:	Tests error message is generated when invalid input path is given
Type:	Unit test (static, automated)
Initial State:	The user has run episode generation
Input:	Invalid input file path
Output:	Error message "Input file is invalid"
Pass:	"Input file is invalid" is outputted in console

Test 6.2.7.3:	Input file invalid error is not generated
Requirements:	R8
Description:	Tests error message is not generated when input path is valid
Type:	Unit test (static, automated)
Initial State:	The user has run episode generation
Input:	Valid input and output file paths and valid input csv
Output:	An output file csv
Pass:	"Input file is invalid" is not outputted in console

Test 6.2.7.4:	Output file path invalid error generated
Requirements:	R14
Description:	Tests error message is generated when invalid output path is given
Type:	Unit test (static, automated)
Initial State:	The user has run episode generation
Input:	Invalid output file path and valid input file path
Output:	Error message "Output file is invalid"
Pass:	"Output file is invalid" is outputted in console

Test 6.2.7.5:	Output file invalid error is not generated
Requirements:	R14
Description:	Tests error message is not generated when output path is valid
Type:	Unit test (static, automated)
Initial State:	The user has run episode generation
Input:	Valid input and output file paths and valid input csv
Output:	An output file csv
Pass:	”Output file is invalid” is not outputted in console and output file is generated

Test 6.2.7.6:	Output CSV data contains correct information
Requirements:	R8, R13, R14
Description:	Tests if output CSV file contains the correct list of stop points and activity locations within radius of the tolerance set by user
Type:	Unit test
Initial State:	User has run episode generation
Input:	Valid input and output file paths and valid input CSV file
Output:	Output CSV generated where user has specified
Pass:	CSV contains all the correct information about each stop point including latitude longitude and the list of activity locations near by

6.2.8 Transformation Module Test Cases

Test 6.2.8.1:	Incorrect file path error message is generated.
Requirements:	NFR 18, 21,26,27
Description:	Tests if only correct file paths are accepted.
Type:	Unit test (static, automated)
Initial State:	The user has ran preprocessing on their initial GPS data set and has ran episodeGeneration.
Input:	Using an incorrect file path for using the stoprelated function of the Transformation module.
Output:	A statement is shown to the user explaining the format of the required filepath.
Pass:	An exception has been thrown

Test 6.2.8.2:	A list of Point type objects representing the stop GPS points are created.
Requirements:	NFR 8,9,11,12,16,17,18,19
Description:	Tests if a csv file containing stop GPS traces are converted into Point objects.
Type:	Unit test (static, automated)
Initial State:	The user has ran preprocessing on their initial GPS data set and has ran episodeGeneration.
Input:	Pass the filepath to the csv containing stop GPS data.
Output:	A list of Point objects containing latitude,longitude, time and mode of travel.
Pass:	A list of Point Objects is returned with the same length of csv data.

Test 6.2.8.3:	A list of Point type objects representing the episode GPS points are created.
Requirements:	NFR 8,9,11,12,16,17,18,19
Description:	Tests if a csv file containing episode GPS traces are converted into Point objects.
Type:	Unit test (static, automated)
Initial State:	The user has ran preprocessing on their initial GPS data set and has ran episodeGeneration.
Input:	Pass the filepath to the csv containing episode GPS data.
Output:	A list of Point objects containing latitude,longitude, time and mode of travel.
Pass:	A list of Point Objects is returned with the same length of csv data.

Test 6.2.8.4:	A list of Point type objects representing the trace GPS points are created.
Requirements:	R17, NFR 8,9,11,12,16,17,18,19
Description:	Tests if a csv file containing trace GPS traces are converted into Point objects.
Type:	Unit test (static, automated)
Initial State:	The user has ran preprocessing on their initial GPS data set and has ran episodeGeneration.
Input:	Pass the filepath to the csv containing episode GPS data.
Output:	A list of Point objects containing latitude,longitude, time and mode of travel.
Pass:	A list of Point Objects is returned with the same length of csv data.

Test 6.2.8.5:	The most frequent mode of travelled is returned.
Requirements:	R16, NFR 8,9,11,12,16,17,18,19
Description:	Given a trace filepath to summary mode function, the most travelled mode of transportation is returned.
Type:	Unit test (static, automated)
Initial State:	The user has ran preprocessing on their initial GPS data set and has ran episodeGeneration.
Input:	Pass the filepath to the csv containing trace GPS data.
Output:	The mode of travelled used the most is returned.
Pass:	A string of mode type is returned.

Test 6.2.8.6:	Incorrect file path error message is generated.
Requirements:	NFR 18, 21,26,27
Description:	Tests if only correct file paths are accepted.
Type:	Unit test (static, automated)
Initial State:	The user has ran preprocessing on their initial GPS data set and has ran episodeGeneration.
Input:	Using an incorrect file path for using the tracerelated function of the Transformation module.
Output:	A statement is shown to the user explaining the format of the required filepath.
Pass:	An exception has been thrown

Test 6.2.8.7:	Incorrect file path error message is generated.
Requirements:	NFR 18, 21,26,27
Description:	Tests if only correct file paths are accepted.
Type:	Unit test (static, automated)
Initial State:	The user has ran preprocessing on their initial GPS data set and has ran episodeGeneration.
Input:	Using an incorrect file path for using the episoderelated function of the Transformation module.
Output:	A statement is shown to the user explaining the format of the required filepath.
Pass:	An exception has been thrown

Test 6.2.8.8:	Incorrect file path error message is generated.
Requirements:	NFR 18, 21,26,27
Description:	Tests if only correct file paths are accepted.
Type:	Unit test (static, automated)
Initial State:	The user has ran preprocessing on their initial GPS data set and has ran episodeGeneration.
Input:	Using an incorrect file path for using the summarymode function of the Transformation module.
Output:	A statement is shown to the user explaining the format of the required filepath.
Pass:	An exception has been thrown

Test 6.2.8.9:	Incorrect input error message for convertActivityLocation(ActvityLoactionList) is generated.
Requirements:	R 8,NFR 18, 21,26,27
Description:	Tests if only correct input which is a list of stop points and their associated activitylocation objects, are accepted.
Type:	Unit test (static, automated)
Initial State:	The user has ran preprocessing on their initial GPS data set and has ran episodeGeneration.
Input:	Using an incorrect file path for using the convertActivityLocation function of the Transformation module.
Output:	A statement is shown to the user explaining the input is incorrect
Pass:	An exception has been thrown

Test 6.2.8.10:	Stop related activity list is returned by convertActivity-Location(ActvityLoactionList).
Requirements:	R 8,13, NFR 8,9,11,12,16,17,18,19
Description:	Tests if the required list containing stop points and their associated attributes are returned.
Type:	Unit test (static, automated)
Initial State:	The user has ran preprocessing on their initial GPS data set and has ran episodeGeneration.
Input:	Using an incorrect file path for using the convertActivityLocation function of the Transformation module.
Output:	list of [point lat, point lon,[nested list of activity locations attributes]] is returned
Pass:	list of [point lat, point lon,[nested list of activity locations attributes]] is returned

Test 6.2.8.11:	Invalid file error for convertActivityCSV is returned.
Requirements:	R8, NFR 18, 21,26,27
Description:	Tests if the required file is an outputted file from fetchActivitylocation
Type:	Unit test (static, automated)
Initial State:	The user has ran preprocessing on their initial GPS data set and has ran episodeGeneration.
Input:	Using an incorrect file path for using the convertActivityCSV function of the Transformation module.
Output:	Invalid file error is returned
Pass:	Exception is thrown

Test 6.2.8.12:	Activity objects in the right format to be used are created by convertActivityCSV
Requirements:	R 8,NFR 8,9,11,12,16,17,18,19
Description:	Tests if the required list of activity objects are generated
Type:	Unit test (static, automated)
Initial State:	The user has ran preprocessing on their initial GPS data set and has ran episodeGeneration.
Input:	the required file, an outputted file from fetchActivitylocation is passed
Pass:	list of activity location objects are returned

Test 6.2.8.13:	Invalid input error for convertListToActivityLocationObject(activityLocationList) is returned.
Requirements:	R 8,NFR 18, 21,26,27
Description:	Tests if the required input is passed
Type:	Unit test (static, automated)
Initial State:	The user has ran preprocessing on their initial GPS data set and has ran episodeGeneration.
Input:	Using an incorrect input for using the convertListToActivityLocationObject function of the Transformation module.
Output:	Invalid input error is returned
Pass:	Exception is thrown

Test 6.2.8.14:	list Activity objects of activity location type are generated.
Requirements:	R 8,NFR 8,9,11,12,16,17,18,19
Description:	activity Location List of required parameters are converted to an activity location class type
Type:	Unit test (static, automated)
Initial State:	The user has ran preprocessing on their initial GPS data set and has ran episodeGeneration.
Input:	Using an activity Location List of required parameters for using the convertListToActivityLocationObject function of the Transformation module.
Output:	list Activity objects of activity location type are generated.
Pass:	list Activity objects of activity location type are generated.

6.2.9 PreProcessing Module Test Cases

Test 6.2.9.1:	Creating processed trace file from raw user trace input.
Requirements:	R1, R2, R3, R4, R17
Description:	Tests if the preprocessing module creates a correctly processed trace file given an input
Initial State:	User inputs csv file with lat,long,time values. Lat and Long are in DD format.
Input:	File path to the input trace CSV output and name of goal directory.
Output:	processed trace csv.
Pass:	goal directory contains a processed trace csv.

Test 6.2.9.2:	Creating processed trace file from DMS lat/long user trace input.
Requirements:	R1, R2, R3, R4
Description:	Tests if the preprocessing module creates a correctly processed trace file given DMS input
Initial State:	User inputs csv file with lat,long,time values. lat and long are in DMS format
Input:	File path to the input trace CSV output and name of goal directory.
Output:	processed trace csv.
Pass:	goal directory contains a processed trace csv with DD lat/long output.

Test 6.2.9.3:	Creating processed trace file from DMS lat/long user trace input.
Requirements:	R1, R2, R3, R4
Description:	Tests if the preprocessing module outputs a processed csv without invalid data points
Initial State:	User inputs csv file with some invalid lat,long,time values.
Input:	File path to the input trace CSV output and name of goal directory.
Output:	processed trace csv.
Pass:	goal directory contains a processed trace csv without invalid data points.

Test 6.2.9.4:	Converting DMS lat/long values to DD
Requirements:	R4
Description:	Tests if DMStoDD function converts DMS values correctly
Initial State:	
Input:	list of DMS latitude and longitude values.
Output:	list of converted DD values.
Pass:	output contains correctly coverted DD values.

Test 6.2.9.5:	Raising invalid input exception
Requirements:	R2
Description:	Tests if invalid input exception is raised with data that does not contain lat,long, and time fields
Initial State:	User has inputted csv file without all of lat,long,time values.
Input:	File path to the invalid input trace CSV output and name of goal directory.
Output:	invalidInputDataException
Pass:	invalidInputDataException raised

6.2.10 Activity Locations Module

Test 6.2.10.1:	Activity Location object is created when parameters are passed to Activity Location constructor
Requirements:	R8,R13,R14
Description:	Tests that ActivityLocation object is able to be created
Type:	Unit test (static, automated)
Initial State:	
Input:	(”Lemon Bar”, 43.651504, -79.386657, ”Juice”)
Output:	ActivityLocation Object
Pass:	Activity Location Object created of type Activity Location

Test 6.2.10.2:	Can fetch ActivityLocation Objects Name
Requirements:	R8,R13,R14,R3
Description:	Tests getting activity location object name
Type:	Unit test (static, automated)
Initial State:	An activity location object has been created
Input:	
Output:	Name of activity location
Pass:	Name of activity location matches name passed as a parameter

Test 6.2.10.3:	Can fetch ActivityLocation Objects Latitude
Requirements:	R8,R13,R14,R3
Description:	Tests getting activity location object latitude
Type:	Unit test (static, automated)
Initial State:	An activity location object has been created
Input:	
Output:	Latitude of activity location
Pass:	Latitude of activity location matches latitude passed as a parameter

Test 6.2.10.4:	Can fetch ActivityLocation Objects Longitude
Requirements:	R8,R13,R14,R3
Description:	Tests getting activity location object longitude
Type:	Unit test (static, automated)
Initial State:	An activity location object has been created
Input:	
Output:	Longitude of activity location
Pass:	Longitude of activity location matches longitude passed as a parameter

Test 6.2.10.5:	Can fetch ActivityLocation Objects Amenity
Requirements:	R8,R13,R14,R3
Description:	Tests getting activity location object amenity
Type:	Unit test (static, automated)
Initial State:	An activity location object has been created with amenity
Input:	
Output:	Amenity of activity location
Pass:	Amenity of activity location matches amenity passed as a parameter

Test 6.2.10.6:	Creating an ActivityLocation Object with default parameters not specified
Requirements:	R8,R13,R14
Description:	Tests if ActivityLocation Object can be created without amenity parameter passed in
Initial State:	An activity location object has been created without amenity
Input:	
Output:	Activity Location Ojbect
Pass:	Amenity of ActivityLocation object is equal to "None"

6.2.11 Point Module

Test 6.2.11.1:	Point object is created when parameters are passed to Point class constructor
Requirements:	R8, R13, R11, R12
Description:	Tests that Point object is able to be created
Type:	Unit test (static, automated)
Initial State:	
Input:	(43.651605, -79.386759,"17:22:02", "mode.DRIVE")
Output:	Point Object
Pass:	Point Object created of type Point

Test 6.2.11.2:	Can fetch Point Objects Latitude
Requirements:	R8,R13,R14, R11, R12
Description:	Tests getting Point object latitude
Type:	Unit test (static, automated)
Initial State:	A Point object has been created
Input:	
Output:	Latitude of Point
Pass:	Latitude of Point matches latitude passed as a parameter

Test 6.2.11.3:	Can fetch Point Objects Longitude
Requirements:	R8,R13,R14, R11, R12
Description:	Tests getting Point object longitude
Type:	Unit test (static, automated)
Initial State:	An Point object has been created
Input:	
Output:	Longitude of Point
Pass:	Longitude of Point matches longitude passed as a parameter

Test 6.2.11.4:	Can fetch Point Object Time
Requirements:	R8,R13,R14, R11, R12
Description:	Tests getting Point Object Time
Type:	Unit test (static, automated)
Initial State:	A Point object has been created with time
Input:	
Output:	Time of Point Object
Pass:	Time of Point matches time passed as a parameter

Test 6.2.11.5:	Can fetch Point Object Mode
Requirements:	R8,R13,R14, R11, R12
Description:	Tests getting Point Object Mode
Type:	Unit test (static, automated)
Initial State:	A Point object has been created with mode
Input:	
Output:	Mode of Point Object
Pass:	Mode of Point matches mode passed as a parameter

Test 6.2.11.6:	Point object is created when default parameters are not passed to Point class constructor
Requirements:	R8, R13, R11, R12
Description:	Tests that Point object is able to be created
Type:	Unit test (static, automated)
Initial State:	
Input:	(43.651605, -79.386759)
Output:	Point Object
Pass:	Point Object created of type Point with point.time = None and point.mode = None

6.3 Tests for Non-Functional Requirements

~~This section will be completed once the MIS has been updated and there is greater clarity on specific modules.~~ This is section will be evaluated manually or be tested through Automated tests such as Stress Testing.

6.4 Traceability Between Test Cases and Modules

This section will be completed once the MIS has been updated and there is greater clarity on specific modules.

References

[1] R. Dalumpines and D. M. Scott, ““gis-based episode reconstruction toolkit (gert): A transferable, modular, and scalable framework for auto-

mated extraction of activity episodes from gps data,”,” Travel Behaviour and Society, 21-Apr-2017., January 2011.

7 Appendix

7.1 Symbolic Parameters

The definition of the test cases will call for SYMBOLIC_CONSTANTS. Their values are defined in this section for easy maintenance.

7.2 Usability Survey Questions?

1. How easy was it to complete tasks using the system?
    (a) Easy
    (b) Neither easy nor difficult
    (c) Difficult
2. How satisfied are you with the overall user experience?
    (a) Satisfied
    (b) Neither satisfied nor dissatisfied
    (c) Dissatisfied
3. Were the instructions clear and easy to understand?
    (a) Clear
    (b) Somewhat clear
    (c) Not at all clear
4. How often did you encounter errors or issues while using the system?
    (a) Never
    (b) Sometimes
    (c) Almost always

Appendix — Reflection

General Team:

To implement the verification and validation plan within our project our team will have to learn a few new skills. The team will have to create a standard testing suite and develop a standard testing method that each member of the team will follow. This will be to ensure that all tests are understandable and readable by all members of the group.

The team will also familiarize themselves with the pytest framework which will allow us to create consistent, efficient tests that will test each function in our program. As well as learn system testing techniques such as —. The team will also need to create a testing strategy that is appropriate and feasible for the project.

Smita Singh:
Smita will be responsible for creating unit test for Route Choice Analysis. She will need understand the inputs and outputs of each of the methods that will be required to perform that specific module. Smita will also be leading the creation of a test strategy.

Moksha Srinivasan:
Similar to Smita, Moksha will also be responsible for creating tests for Route Choice Analysis. Moksha will also be responsible for helping set up the standard test suite and implementing CI/CD into our git repository by following the tutorial given by Chris Shankula.

Longwei Ye:
Longwei will be responsible for creating unit testing for trip trajectory. Longwei will be responsible for learning about best testing practises through research and will ensure that the team sticks to those practises.

Niyatha Rangarajan:
Niyatha will be creating unit testing module for travel episode verification and categorization. Niyatha has been passionate about end to end system testing. She will be researching how to perform relevant system testing by researching industry standards and then be responsible for informing the rest

of the team.

Abeer Alyasiri:
Abeer will be responsible for learning about different file formats (CSV, XML, JSON, SHP) and the most efficient ways of parsing through and transforming data. This will ensure that modules are well designed and time efficient. She will ensure test cases include all relevant input file formats and malformed data inputs as well. She will learn about these best practices through the use of data parsing python tutorials online as well as researching libraries/prior implementations of open source GIS analysis tools. Through her co-op position, Abeer has significant experience working with various types of data and hence is the most suited member of our team for this task.

Nicholas Lobo:
Nicholas has always been interested in learning about data analysis and normalization. Within the scope of this project, he has taken on the responsibility of learning about GPS data standards to help provide a wide range of inputs for all tests. He will ensure that the preprocessing unit can handle various types of GPS data as well as inform decisions about edge cases related to incorrect data. He can complete this task by consulting academic papers that detail the characteristics of and how to parse GPS data.