0.1 Test cases

Item	Description
Name	Command line interface (CLI) functionality
Test identifier	UNIT-01
Person responsible	Henrik Knutsen
Feature(s) to be tested	That all possible commands are working correctly when using the CLI. That input with incompatible commands does not run.
Pre-conditions	Code for input handling for all possible commands. Code for error handling for invalid inputs.
Execution steps	1. Start the program with all possible commands. Run once for each command. 2. Start the program with all possible combinations of incompatible commands. Run once for each combination of incompatible commands.
Expected result	1. The program starts successfully, performing the action connected to the command. 2. The program aborts startup. Gives error warning: Invalid arguments.

Item	Description
Name	P3P parser
Test identifier	UNIT-02
Person responsible	Henrik Knutsen
Feature(s) to be tested	That the P3P parser correctly parses all the required fields and their values.
Pre-conditions	The P3P parser must be implemented. Need to have P3P policies with a wide range of cases.
Execution steps	1. Manually go through a P3P XML and obtain all the required fields and their values. 2. Run the same P3P XML in the P3P parser and print the parsed elements and their values to console. 3. Compare the results from the two parsing methods.
Expected result	1. All required fields and values are obtained and written down. 2. The P3P XML is parsed successfully. Its content is printed to console. 3. The results are identical. The the two outputs on paper and in console have the same fields, with the same values.

Item	Description
Name	Local database
Test identifier	UNIT-03
Person responsible	Henrik Knutsen
Feature(s) to be tested	Writing to and reading from the local database. That the serialization of the database is working.
Pre-conditions	Code for writing to and reading from the database file. Need to have two different P3P policies.
Execution steps	1. Write policy A to the local database. 2. Write policy B to the local database. 3. Read policy A from the local database. 4. Read policy B from the local database. 5. Compare the written policy A and the read policy A. 6. Compare the written policy B and the read policy B.
Expected result	1. Policy A is successfully written to the database file. 2. Policy B is successfully written to the database file. 3. Policy A is successfully read from the database file. 4. Policy B is successfully read from the database file. 5. The written policy A and the read policy A are identical. They both have the same fields, with the same values. 6. The written policy B and the read policy B are identical. They both have the same fields, with the same values.

Item	Description
Name	Graphical user interface (GUI) functionality
Test identifier	UNIT-04
Person responsible	Henrik Knutsen
Feature(s) to be tested	That all the elements - buttons, lists etc. - are working as intended.
Pre-conditions	GUI with all the necessary listeners must be implemented. Code for running the program with the GUI.
Execution steps	1. Start the program using the GUI. 2. Use all the elements in the GUI.
Expected result	1. The program starts successfully in GUI mode. 2. All the elements are triggering the connected methods when used.

Item	Description
Name	Algorithm classification
Test identifier	UNIT-05
Person responsible	Henrik Knutsen, Dimitry Kongevold
Feature(s) to be tested	That the k-nearest neighbor algorithm bases its decision on the k most similar policies
Pre-conditions	Code for reading from the weights file must be implemented. A working k-nearest neighbor algorithm that uses the weights must be implemented. Need one policy to test on, and a set of policies to be used as history.
Execution steps	1. Load a set of policies into the history. 2. Run the distance algorithm on the single policy and the history. 3. Manually calculate and write down the distances between the single policy and each of the policies in the history. 4. Compare the distances that are returned by the algorithm with the manually calculated distances. 5. Run the reduction algorithm with input to find the k nearest policies. 6. Manually find the k policies with the lowest distance. 7. Compare the policies returned by the reduction algorithm and those found manually. 8. Run the conclusion algorithm.
Expected result	1. The policies are successfully stored in the database. 2. The distance algorithm runs successfully for the chosen input, and returns the distances between the single policy and each of the policies in the history. 3. The distance between the single policy and the policies in the database are found and written down. 4. The distances returned by the algorithm, and the respective distances calculated manually are identical. 5. The reduction algorithm runs successfully for the chosen input and returns the k nearest policies. 6. The k nearest policies are found and written down. 7. The k policies returned by the reduction algorithm and the k policies found manually are the same policies. 8. The conclusion algorithm runs successfully.

Item	Description
Name	Algorithm learning
Test identifier	UNIT-06
Person responsible	Henrik Knutsen, Neshahavan Karunakaran
Feature(s) to be tested	That the weights file is updated when a new policy is added to history.
Pre-conditions	Code for reading from and writing to the weights file. Code for writing to the database. Algorithms for classification and learning must be implemented.
Execution steps	1. Obtain and write down the contents of the weights file. 2. Load a set of policies into the history. 3. Run the classification algorithm on the single policy and the history. 4. Accept the recommendation. 5. Choose to save the new policy to history. 6. Obtain and write down the contents of the weights file. 7. Compare the contents of the weights files obtained in steps 1. and 6.
Expected result	1. The contents of the weights file is obtained and written down. 2. All the chosen policies are successfully stored in the database. 3. The algorithm classifies the new policy, gives a recommendation on what action to take, and asks the user if the recommendation is accepted. 4. The user is asked whether to save the new policy to history or not. 5. The new policy is successfully written to the database. 6. The contents of the weights file is obtained and written down. 7. At least one of the fields in the weights files obtained from steps 1. and 6. are different.

Item	Description
Name	Packet passing through network to community database
Test identifier	UNIT-07
Person responsible	Henrik Knutsen
Feature(s) to be tested	That packets can be sent between the client program and the community database.
Pre-conditions	A running local client. A (virtual) server. Code for sending and receiving packets must be implemented.
Execution steps	1. Start the program locally. 2. Start the (virtual) server. 3. Send policy A from the local client to the server. Print the contents of policy A. 4. Receive packet A at the (virtual) server. Print the contents of policy A. 5. Compare the sent policy A and the received policy A. 6. Send policy B from the (virtual) server to the client. Print the contents of policy B. 7. Receive policy B at the local client. Print the contents of policy B. 8. Compare the sent policy B and the received policy B.
Expected result	1. The program starts successfully. 2. The (virtual) server starts successfully. 3. Policy A is sent to the server. Its contents are printed to console. 4. Policy A is received at the (virtual) server. Its contents are printed to console. 5. The contents of the sent policy A and the received policy A are identical. 6. Policy B is sent to the local client. Its contents are printed to console. 7. Policy B is received at the local client. Its contents are printed to console. 8. The contents of the sent policy B and the received policy B are identical.