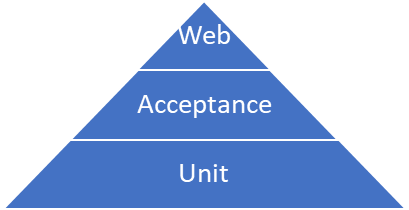
**6 Testing**

**6.1 System Testing Approach**

Testing was required for the system to ensure changes to the code base maintain proper system functionality. An effective software testing approach enables developers to make changes with confidence and ensures quality code that meets stakeholder requirements [2]. In **Figure 6.1** the three types of testing are in a hierarchal structure because the higher-level the testing is the fewer tests there are. Because if a higher-level test uncovers an error in the code base and no lower-level test reports a failure, a lower level test needs to be written that fails until the error is fixed. This approach ensured that the tests follow the DRY (Don’t Repeat Yourself) software development principle by not overlapping tests testing the same case at different levels. Unit tests are also automated and modern machines can run thousands of unit tests within a few minutes [4]. This is not possible with the higher-level tests in the system as they require manual human interaction, and hence there is less tests at that level.



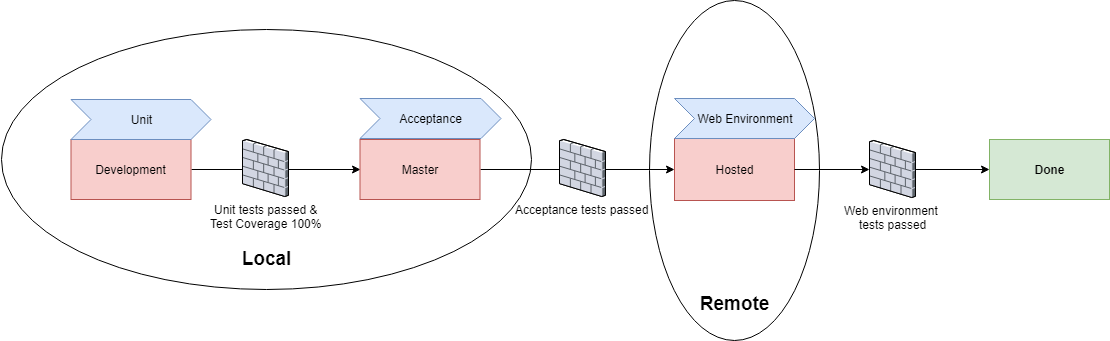
**Figure 6.1** Testing pyramid with the layers of testing for the system.

The testing levels test different granular parts of the system:

* Unit tests: ensure individual atomic functions return the expected results.
* Acceptance tests: ensure whether the acceptance criteria of functional and non-functional requirements identified in **chapter 3.4** are met by following the user flow in **chapter 3.2**.
* Web environment tests: ensures the correct functioning of the system in a web environment and assesses whether the performance is acceptable.

There are no integration tests as there are no integration with external systems.

At the completion of the implementation on the development feature branch a user story is moved to the ‘Testing’ phase, **see chapter 3.5**. Whenever all the user stories of a feature are completed the changes are merged with master. Testing levels occur at different stages of source control integration and to progress to the next stage there is testing quality gate requirements to ensure the feature is fully tested **see Figure 6.2.** Only once all levels of testing have been passed the feature is ‘done’.



**Figure 6.2** The stages of testing for a feature that is implementation complete. The source control of branches is in red, with the testing type occurring at that source control integration stage. The quality gate requirements that must be met before progressing to the next stage are visible.

**6.2 Unit Testing**

**6.2.1 Methodology**

Scope

There were decisions made on what functions to write unit tests for in the system. Unit test class should generally test callable public methods [2]. However, as the OOP design principle was not implemented there is no function encapsulation, and hence all functions in the solution are publicly callable by importing their module. To save time writing tests for every function, unit tests were written for functions that contained several internal calls to test they were functioning correctly together. Callback functions were not tested as their correct functioning was verified through manual acceptance testing.

The following back-end functions were tested:

* Functions that are called by a callback function
* Internal functions with conditional cases
* Functions called by multiple callers e.g. shared functions for generating graph elements.

Test Structure

Unit tests were written to test positive (value as expected) and negative (error) cases, testing the expected return value of a single function i.e. “the unit” by passing different arguments. The unit tests were written in the format:

* Arrange the required data to use in the parameters of the function the unit tests calls
* Act by calling the function
* Assert the return value of the function is equal to the expected return value

Test Driven Development

Unit tests were written using a test driven development (TDD) approach. Tests were written before the functionality was implemented that fail. Functionality was then implemented to pass the tests. This ensured that functionality included was fully tested that enables 100% coverage in the coverage metric in **chapter 6.2.2.** In addition, it ensures functionality meets the defined requirements of the system, requiring only to have one single responsibility for the system. This is preferred than being written in a monolithic pattern with multiple responsibilities, which makers the code more difficult to understand [3].

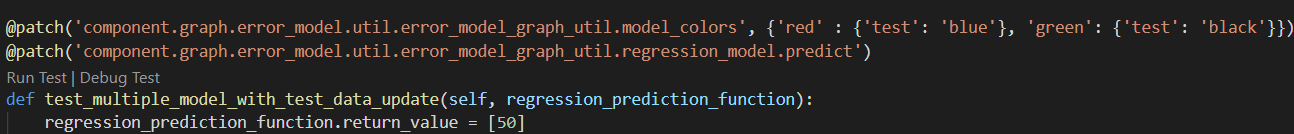
**6.2.2 Implementation**

File and Folder Organisation

The unit test files are contained within the *test* folder and the folder structure is the same as the implementation structure. The files are named with the implementation file being tested and ‘\_test’ appended. Test case functions are named with ‘test\_’ at the start to inform the test runner about which methods in the test classes are individual tests.

Testing Library

The **unittest** Python library was used to create a test suite with a collection of test cases for a single function. The tests are then executed using a command-line test runner, which states the outcome of the tests. The test themselves are encapsulated in a TestCase class, making them self-contained with no dependency on other unit tests.

Mocking objects and functionsThe python **mock** library was used to patch functions with ‘faked’ return values. This saved time passing correct arguments for a function that were passed onto an internal function that would return an unexpected value if called with incorrect arguments. The mock library was also used to mock objects that the function model imports to reduce external dependencies and make the test verification more manageable. In **Figure 6.3**, the model predict functionality were mocked to return dummy load values when testing the update methods of the graph components as the Scikit-learn functionality would return an exception with invalid arguments. The list mapping of a model colour to a test colour was patched to make the mapping functionality working easier to verify and eliminate a dependency on the imported module with the list of model colours.

**Figure 6.3** An example of the mock library used in a test case to mock module imported objects and function return values

Test Coverage

To ensure all conditional paths within the functions tested were followed by the unit test cases the Python library **coverage** was used to generate a code coverage report. The report detailed the % of code followed by the unit tests. If a file had a low % coverage, new unit tests were written based off an HTML report highlighting what paths in the code were not being followed. The new unit tests followed the paths identified as not being taken by the report. The code coverage report acted as a quality gate in development. All functions capturable by unit tests needed to have 100% coverage or close (circumstances permitting) before the code changes were merged with the development branch. This provided confidence that the current state of the system being developed did not have broken functionality. A report detailing the coverage of each file in the system is included in the **appendices**.

Automation

**Visual Studio Code** was used as the IDE for developing the system. The IDE provided the automatic discovery, running and integrated terminal for success output of unit tests. Configuration settings specific to the system are contained in the *.vscode* folder with the unit test implementation (unittest) being set and various command line arguments passed to the command to run unit tests e.g. verbose output for debugging. This enabled responsive feedback on the tests when making code changes, instead of spending time remembering the manual commands for running unit tests. *\_\_init\_\_.py* files in each test folder structure were required to make the test files packaged, and hence discoverable by the unittest runner [4]. These files were hidden using a regular expression in the configuration settings. The folder structure in the IDE only contains test files, hiding redundant files not relevant to writing unit test cases. The configuration settings are shared within the git repository, providing a common test environment when using a new environment to develop the system.

**6.3 Acceptance Testing**

**6.3.1 Methodology**

Manual vs Automated Testing

Automation is the preferred approach to writing acceptance tests. Manual testing requires a person to manually go through a test script and verifying the system output for each case, which takes longer and is more prone to errors than a computer [5]. Early in the system’s development the testing framework **Robot** and **Selenium** was used to write an automated test to verify the solution’s graphical presentations displayed the correct range of load data in a web browsers. However, there was no open source library written for Robot to interact with Plotly graphs and gather data from them to verify their presentation in the system. It was determined that the effort to write these libraries was not as valuable to the stakeholder as the features being developed. Automated test scripts are not very effective at verifying the non-functional requirements of the user e.g. the usability and styled appearance of the system [2]. Given the increased development work required to write libraries to facilitate the automation of acceptance tests, and as the system is focused on delivering visual results to the user, acceptance testing was performed manually using a test script.

Scope

Acceptance tests replicate a user’s flows through the program, black-box testing the core functionality of the system. The tests are visual, validating how graphs and tables are presented after system components are interacted with in the defined order and provided parameters. Unlike the unit tests, they do not test the intermediate functions used to generate the changes to the UI state [5]. The test environment for the acceptance tests is the user’s own local machine, with the setup of this being included in the first part of the acceptance tests.

**6.2.2 Implementation**

External Dependencies

Test data is used as the source of data for the acceptance tests to enable the tester to test both the preprocessing of a dataset and then use the dataset to validate visual features of the solution. It provides a consistent range of data for the user to modify in the components e.g. the date picker, and the same output results to validate every time the acceptance tests are followed. Each part of the test script was written to be independent, only having a dependency on part 1. If there is limited time only tests validating the area of implementation are ran to provide confidence in the system is functioning correctly.

Cross-Browser Testing

The acceptance test script was followed using the three most popular Windows web browsers by market share [6]: Google Chrome, Mozilla Firefox and Internet Explorer. This cross-browser testing ensures that the system functionality works, is performant and is presented with no visual irregularities. Any errors identified are fixed to ensure a consistent user experience across the popular web browsers. Other operating systems and mobile devices were not tested as the application was not deployed locally to these environments during development.

Quality Control Measure

The test script used for the acceptance tests was updated each time a new feature was implemented onto the feature development branch to ensure it covered its end to end functionality. It is assumed that code in the master branch is production ready [2]. Hence, following the acceptance test script was a prerequisite before merging the code base into the master production branch. It provided confidence the solution provided the functionality desired by target users.

**6.3.3 Web Environment Testing**

**6.3.1 Methodology**

Scope

Acceptance tests are limited in that they do not test the system up and running in a web environment, only in a closed local environment. Testing how the system performs in a web environment was important as this is a user’s perspective when the system is deployed to a remote web server. In web environment testing the same test script as acceptance testing in the **appendices** is followed on the web hosted system. Exploratory testing was also performed at this level to uncover errors not identified in lower levels because of the different host environment. Errors identified at this level were documented and the required implementation fixes were made on the master branch.

Proof of Concept

Hosting the system on the internet provided a proof of concept to the stakeholder that the application was deployable to web infrastructure and can serve multiple remote users. The stakeholder and parties with an interest in the system could also access the system remotely rather than having to setup the solution locally, which requires setup time and technical expertise. The multiplatform usage of the system by users identified in **chapter 4.3.1** was also tested at this level, with the presentation of components being tested on devices with variable screen size, modes of interaction and hardware specifications to ensure the system presents and provides core functionality correctly. he functionality hosted is from the master branch, which has been tested locally by unit and acceptance tests to ensure it provides the correct output to the user.

Performance Considerations

Slow response times interacting with the components in web environment that effect the usability of the system were of a specific focus. The remote web server response time to requests is a variable not present in the lower-level testing, which only interfaces with the local machine’s local hosted server. The concurrent use of the solution in the dataset over a web browser was tested identified in **chapter 4.1.2**, with multiple users not on the same network interacting with the solution transforming the same master load dataset to produce results.

**6.3.2 Implementation**

Web Platform

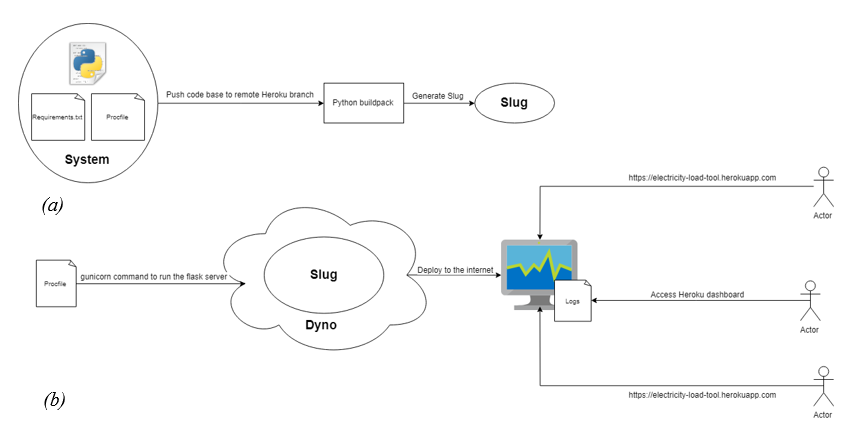
The cloud platform **Heroku** was chosen to host the system on the internet. Heroku facilitates the deployment, execution and management of applications written in many different languages. The reasons for choosing Heroku to host the infrastructure for the system were as follows:

* It is a PaaS (platform as a service), with the operating system and hardware the system hosted on being abstracted. Hence, full focus for deploying the application was on the application.
* It has a free plan with adequate hosting hours (1000) to accommodate the testing usage of the hosted system.
* The hardware provided for the system (512MB RAM and 500MB disk space) exceeds the file size of the include load dataset (52mb) and installed Python libraries on the system.
* There are add-ons that facilitate the building of the application and the integration of access control.
* The deployment process of Heroku is through pushing the code base using Git to a remote Heroku branch. This is the same process as pushing changes to the development and master code branch, and hence seamlessly integrated with the source control tools used during development [7].

Deployment

Heroku made the deployment of the system to the cloud platform seamless as there are no external dependencies, only the system. The system’s code base and the library dependencies are packaged into a ‘slug’ ran by the build pack setup in the Heroku dashboard, see **Figure 6.4(a)** . Initially, the Anaconda build pack was used to generate the system slug. Anaconda was used for package management in the implementation, **see chapter 5.1,** and hence the *environment.yml* file was used to create the slug with the library dependencies required for the system. However, the slug generated by the Anaconda build pack exceeded the maximum slug size (500MB). Hence, the Python build pack was used with with the required library dependencies listed in a *requirements.txt* file in the root of the code base to be installed by the Python package manager **pip**. A drawback to this was that both the *environment.yml* and *requirements*.*txt* had to be updated when the library dependencies of the system changed to ensure the system compiles and functions correctly in both the local and remote environment.

A dyno, an isolated virtualised container that provides the environment for the slug to execute is preloaded with the slug, **see Figure 6.4(b).** Multiple dynos can be preloaded to make the system scalable in terms of hardware resources providing functionality [7]. Commands that Heroku uses to execute the dyno are contained within the *Procfile* in the root folder of the code base. **Gunicorn** server was called to extract the Flask server in *run.py* - the entry point of the system and execute the program. When the system is deployed time-stamped logs are generated from the Flask web server which can be viewed using the Heroku dashboard for debugging errors with the server responses to user requests, **see Figure 6.5.**



**Figure 6.4** The processes to host the system online using Heroku. (a) The creation of a slug that bundles all the dependencies required to run the system in one package. (b) The deployment of the system by executing the slug on the host dyno using the gunicorn web server command in the Procfile. The user’s methods to access the web hosted system for testing and logs for debugging are visualised.



**Figure 6.5** Example of system logs viewed in the Heroku dashboard.

**Citations**

[1]<https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Cross_browser_testing/Introduction>

[2] <https://martinfowler.com/articles/practical-test-pyramid.html>

[3] <https://www.guru99.com/test-driven-development.html>

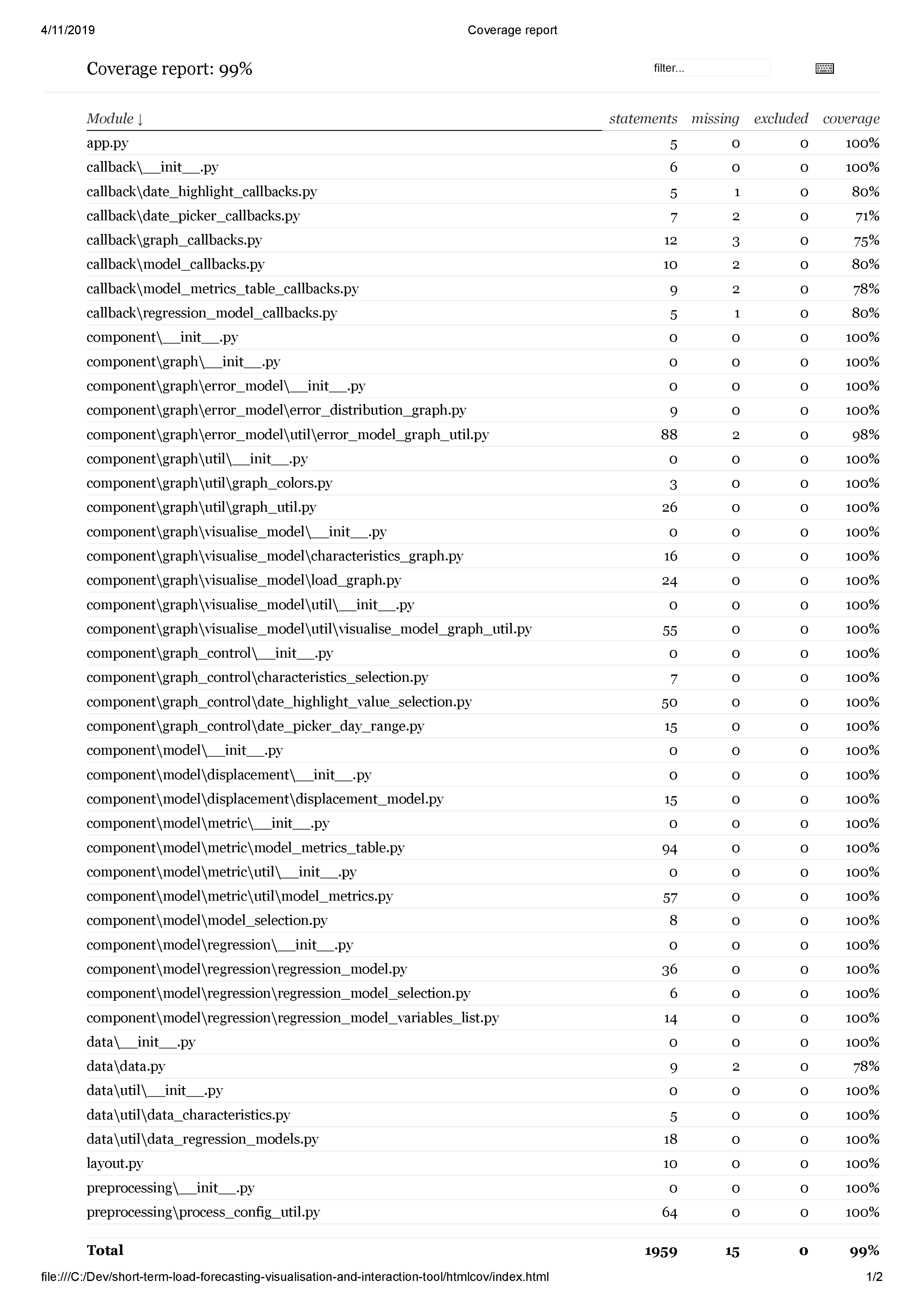
[4] <https://docs.python.org/3/library/unittest.html>

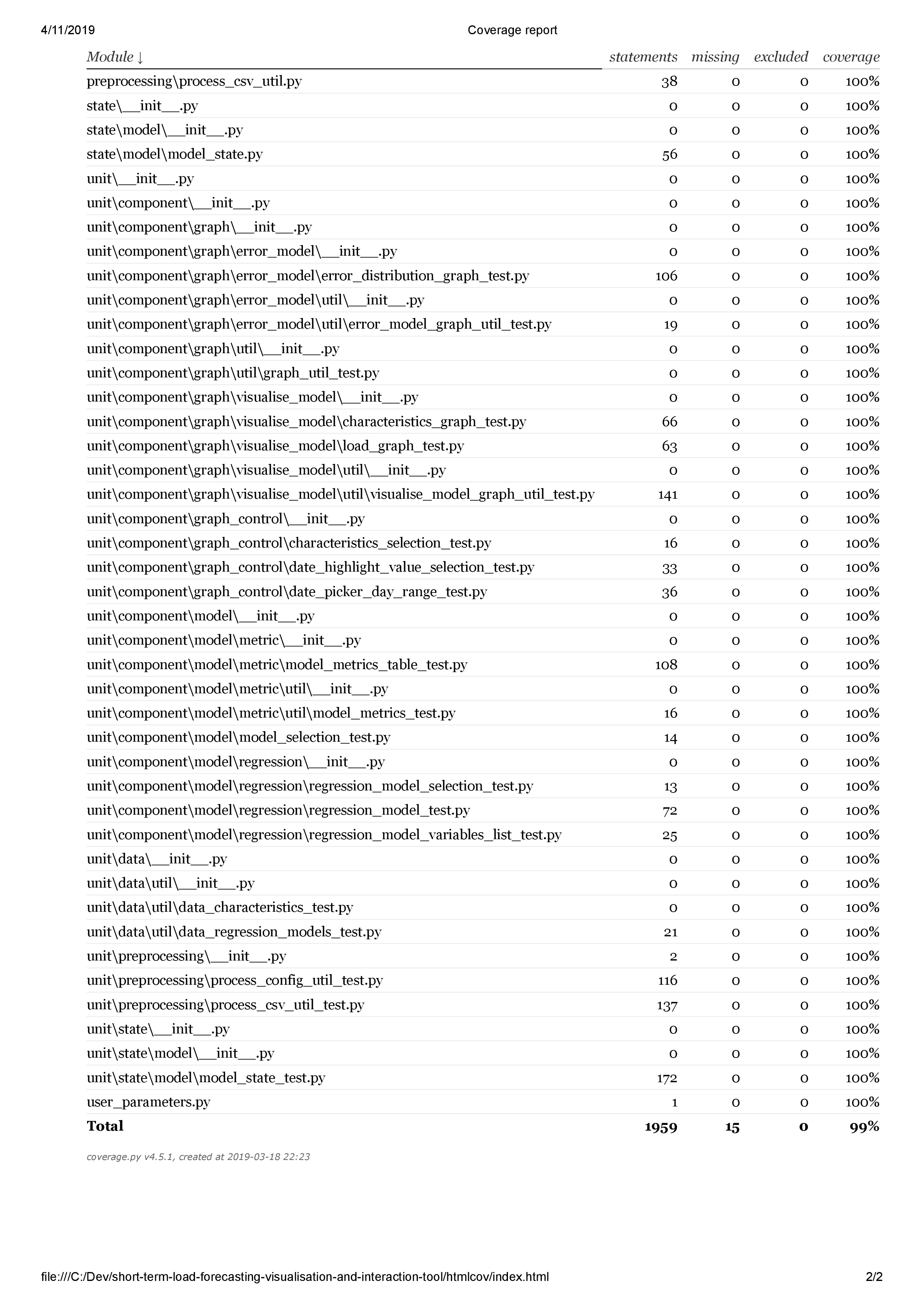
[5] <https://www.atlassian.com/continuous-delivery/software-testing/types-of-software-testing>

[6] <https://www.w3counter.com/globalstats.php>

[7] https://devcenter.heroku.com/articles/how-heroku-works#defining-an-application

**Appendices**





**Part 1: Test environment setup**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Actions** | **Expected Result** | **Done** |
| 1 | Copy file ‘Test Data.csv’ in the test directory and paste to:  *<project-install-directory>\short-term-load-forecasting-visualisation-and-interActions-tool\preprocessing* |  |  |
| 2 | Execute *python process\_csv.py.* |  |  |
| 3 | Type in:   * *Test Data* * *NorthernIreland* * *Press enter twice to skip state and province* |  |  |
| 4 | Delete ‘Test Data.CSV’ in *preprocessing*.  Rename ‘Test Data\_processed.csv’ to ‘SONI.csv’  Cut ‘SONI.csv’ and paste to:  *<project-install-directory>\short-term-load-forecasting-visualisation-and-interActions-tool\data\load\_data*  If SONI.csv ‘already exists rename the existing SONI.csv in the *load\_data* folder to ‘SONI\_original.CSV’ and then paste |  |  |
| 5 | In the directory  *<project-install-directory>\short-term-load-forecasting-visualisation-and-interActions-tool*  Execute *python run.py* |  |  |
| 6 | Using a web browser, navigate to web address *http://127.0.0.1:8050/* | * No graph visible |  |

**Part 2: Load Data Visualisation**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Actions** | **Expected Result** | **Done** |
| 1 | Change start date to Monday 8th February 2010 | * No graph visible |  |
| 2 | Change start date to Monday 8th February 2010.  Change end date to Tuesday 9th February 2010 using the visualised calendar widget | * X axis labelled ‘Date’ * Y axis labelled ‘Load’ * No legend visible * Graph plotted with start date midnight point plotted (2010-02-08) and end date (2010-02-09) midnight point plotted * X axis shows the single day range (3hr increments) * No model metrics table visible below graph |  |
| 3 | Click the ‘+’ labelled button beside the calendar date selection widget | * Start date in the calendar widget is 2010-02-09 and end date is 2010-02-10. * Graph plotted with start date midnight point (2010-02-09) plotted and end date midnight point (2010-02-10) plotted |  |
| 4 | Click the ‘-’ labelled button beside the calendar date selection widget | * Start date in the calendar widget is 2010-02-08 and end date is 2010-02-09. * Graph plotted with start date midnight point (2010-02-08) plotted and end date midnight point (2010-02-09) plotted |  |

**Part 3: Forecasting Visualisations and Statistical Metrics**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Actions** | **Expected Result** | **Done** |
| 1 | Change start date to Tuesday 9th February 2010.  Change end date to Wednesday 10th February 2010.  Open the displacement model carat.  Type in 1 to displaced by text field and select ‘Days’ in dropdown.  Click ‘Add Model’ button. | * Directly below 2010-02-09 18:00’ actual load point (blue).   there is a ‘2010-02-08 18:00’ -1 days point (orange).   * ‘Actual Load’ and ‘-1 Days’ legends visible. * Metrics table has a ‘Visualised’ column. * No ‘Test Data’ sub column below -1 Days column |  |
| 2 | Change test data start date to 8th February 2010. | * No ‘Test Data’ column |  |
| 3 | Change test data start date to 10th February 2010.  Change test data end date to 11h February 2010. | * ‘Test Data’ column |  |
| 4 | Type in 1 to displaced by text field and select ‘Hours’ in dropdown.  Click ‘Add Model’ button. | * Directly below 2010-02-09 18:00’ actual load point (blue).   there is a ‘2010-02-08 18:00’ -1 days point (orange) and ‘2010-02-09 17:00’ -1 hours point (green)   * ‘Actual Load’, ‘-1 Days’ and ‘-1 Hours’ legends visible. * Metrics table has a ‘-1 Hours’ column with ‘Visualised’ and ‘Test Data’ sub columns |  |
| 5 | From models visualised dropdown select ‘-1 Hours’  Click trash icon. | * No -1 Hours legend * No -1 Hours green points * No -1 Hours selection in models visualised dropdown * Metrics table -1 Hours column is not visible. |  |
| 6 | From models visualised dropdown select ‘-1 Days’  Click trash icon. | * No -1 Days legend * No -1 Days points * ‘No Results Found’ in models visualised dropdown * Metrics table is not visible. |  |

**Part 4: Forecasting Model Characteristics**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Actions** | **Expected Result** | **Done** |
| 1 | Change start date to Monday 9th February 2010.  Change end date to Tuesday 10th February 2010.  Type in 1 to displaced by text field and select ‘Days’ in dropdown.  Click ‘Add Model’ button.  Choose ‘Temperature’ in the characteristic graph y-axis dropdown | * -1 Days legend on both graphs. * Colours for each plot (Actual Load and -1 Days) are the same in the two visualised graphs. * Gridlines line up vertically. * Model visualisation (top) graph has a Load labelled y axis and does not have a labelled x axis. * Characteristics graph has a Temperature labelled y axis and a Date labelled x axis. |  |
| 2 | Choose ‘None’ in the characteristic graph y-axis dropdown | * Characteristics graph is not visible * Graph has returned to its original height. |  |

**Part 5: Highlighting Data Points**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Actions** | **Expected Result** | **Done** |
| 1 | Choose ‘Temperature’ in the characteristic graph y-axis dropdown  Choose ‘Day’ and ‘Monday’ in highlight points dropdowns | * No additional plots on the two visualised graphs. * ‘Highlighted Load’ legend added to both graphs |  |
| 2 | Choose ‘Day’ and ‘Tuesday’ in highlight points dropdowns | * ‘Highlighted Load’ legend added to both graphs * Red line plot superimposing over all 2010-02-09 day (blue) points. * No red line plot superimposing -1 Day model (orange) points. |  |

**Part 6: Model Error Distribution Graph**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Actions** | **Expected Result** | **Done** |
| 1 | Click on ‘Models Error Distribution’ | * No graph visualised |  |
| 2 | Change start date to Tuesday 9th February 2010.  Change end date to Wednesday 10th February 2010.  Type in 1 to displaced by text field and select ‘Days’ in dropdown.  Click ‘Add Model’ button. | * Model error distribution graph is visible * -1 Day (Visualised) legend is visible * Y axis labelled ‘Cumulative Percentage (%)’ * X axis labelled ‘Absolute Percentage Error (%)’ * Range of Y axis 0-100 * ‘-1 Days’ colour (orange) is the same in the model visualisation and characteristics graph |  |
| 3 | Type in 1 to displaced by text field and select ‘Hours’ in dropdown.  Click ‘Add Model’ button. | * Model error distribution graph is visible * -1 Day and -1 Hours legend is visible * Range of Y axis 0-100 * ‘-1 Days’ colour (orange) and ‘-1 Hours’ colour (green) is the same in the model visualisation and characteristics graph |  |
| 4 | Choose ‘Day’ and ‘Monday’ in highlight points dropdowns | * No ‘Highlighted’ legend * No red highlighted point plots on graph |  |
| 5 | Choose ‘Day’ and ‘Tuesday’ in highlight point dropdowns | * Red highlight points on superimposing -1 Hours and -1 Days points * ‘-1 Days (Visualised)’, ‘-1 Hours (Visualised)’ and ‘Highlighted’ legends |  |
| 6 | From models visualised dropdown select ‘-1 Days’  Click trash icon. | * ‘-1 Days (Visualised)’, ‘Highlighted’ legends * ‘-1 Days’ orange plots no longer visible * ‘-1 Hours’ point and line colour (green) is the same in the model visualisation graph. |  |
| 7 | Change test data start date to 10th February 2010.  Change test data end date to 11h February 2010. | * ‘-1 Hours (Visualised), ‘-1 Hours (Test) and ‘Highlighted’ legends * Test data plots (Blue-green) plots visible * Test data points not superimposed by any red highlight points |  |
| 8 | From models visualised dropdown select ‘-1 Hours’  Click trash icon. | * Graph is not visible |  |