eMot

Testing Report

Michael Savage - 17313526

Gerard Slowey - 17349433

Supervisor: Jennifer Foster

Date Completed: 02/05/2021

Table of Contents

Testing Overview

This document outlines the testing strategy carried out during the development of the eMot program. Testing was conducted on all components that make up the eMot system to ensure a high quality system with proper execution behaviour.

Tests run on the system will be documented here along with the expected result and actual result and test outcome.

Testing Environment

- Windows 10 64-bit
- Python v3.9.4
- Pytest v6.2.3

Testing Strategy

The system testing strategy was split into two sections, front end GUI testing and backend logic tests covering the program logic and emotion classification.

Front End GUI tests

Because of the application's heavy integration to the PyQt GUI library, integration tests were difficult to define. Furthermore due to the changing graphical layout of the application towards the end of the project development, previously defined GUI integration was no longer applicable. To combat this we focused on manual testing and user testing.

- Ad hoc testing was carried out during the design phase of the GUI using the PyQt designer, such as ensuring the correct labelling of buttons and menus for reference purposes.
- Manual testing was carried out to ensure that buttons executed the correct functions and that graphing data was correctly displayed with matching labels.
- User testing was used to collect feedback on the usability of the system and to get an insight into the user experiences using the system.

Backend Tests

Since the system is a locally executing program all tests were run on the local application. Our application is written entirely in Python so we decided to use the Pytest unit testing library.

A test driven approach to development was taken during this project, with unit tests being defined before code was written. As tests were written first, we found it easier to program logic as we had questioned assumptions of execution behaviour.

During the development process adhoc testing was used to reduce debugging work along the way. Additionally to functionality tests of system logic, classifier functionality and performance profiling were completed.

The trained emotion classifiers were tested for accuracy and expected results as well as profiled for execution performance.

Testing Automation

Automated tests were integrated using the Gitlab runner CI pipeline. The pipeline consisted of two stages with stage one being static code linting using flake8 and stage 2 execution of tests using the Pytest testing framework.

Git Hooks

We integrated 20 pre-commit client side hooks to check various files across the project in the docs, res, and src folder. This ensured that the code was very maintained and easy to read while reducing commit errors and rework.

The git hooks were especially helpful for automatically formatting the converted Python files from QT Designer. These files were messy conversions and they were hard to follow as they were never meant to be edited but instead inherited into a super class. For example, the autopep8 pre-commit was able to make the files conform to the PEP 8 style guide when trying to git commit. The commit could not be pushed to the repo unless all the tests passed.

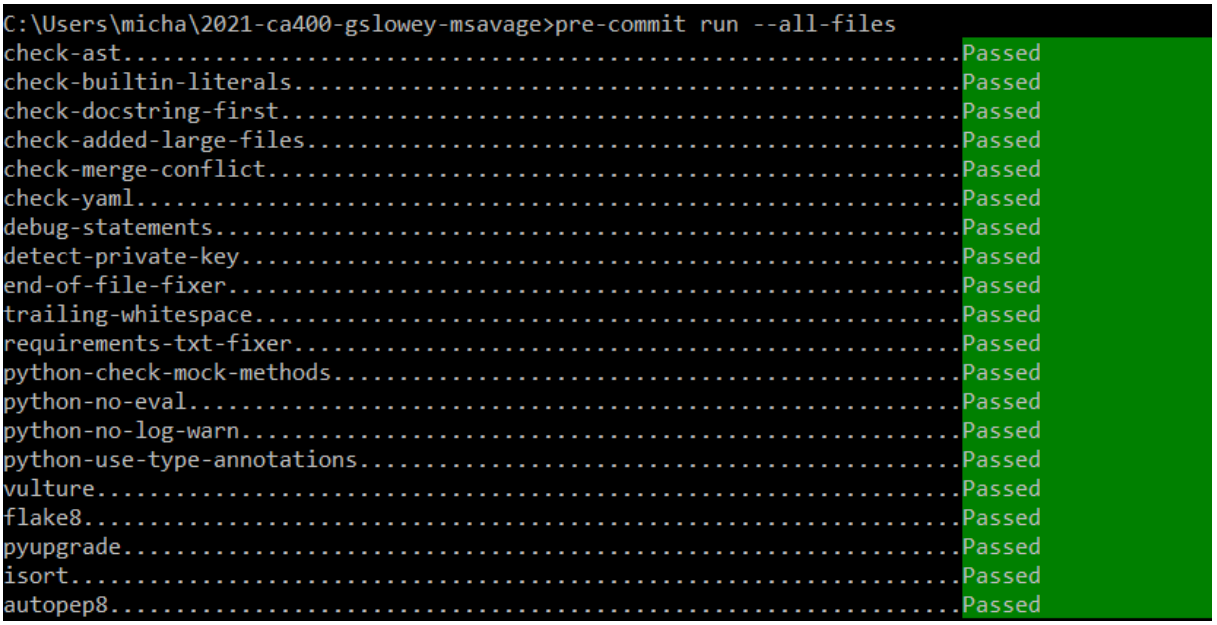

Figure 1: Manual run of pre-commit showing tests passing

Results

Unit Testing

As stated we used the Pytest testing framework to execute unit tests for our project. Unit tests could be run locally as well as always been run by the gitlab CI runner. The automated tests behaviour was defined in the gitlab-ci.yml file. These automated tests highlighted errors on multiple occurances. After installing the project requirements, the Gitlab Runner would run 35 different pytests.

These tests covered the functionality of:

- The blacklist filter - asserting expected filters and base URL checks.
- The browsers - using pytest Monkey Patch to mock environments.
- Retrieving history - asserting that parameters work and asserting error messages.
- Modifying json blacklist - checking duplicates and removing nonexistent items.
- Pre-processing text - asserting expected text after processing.
- Emotion classification - asserting that sentences are being classified suitably.

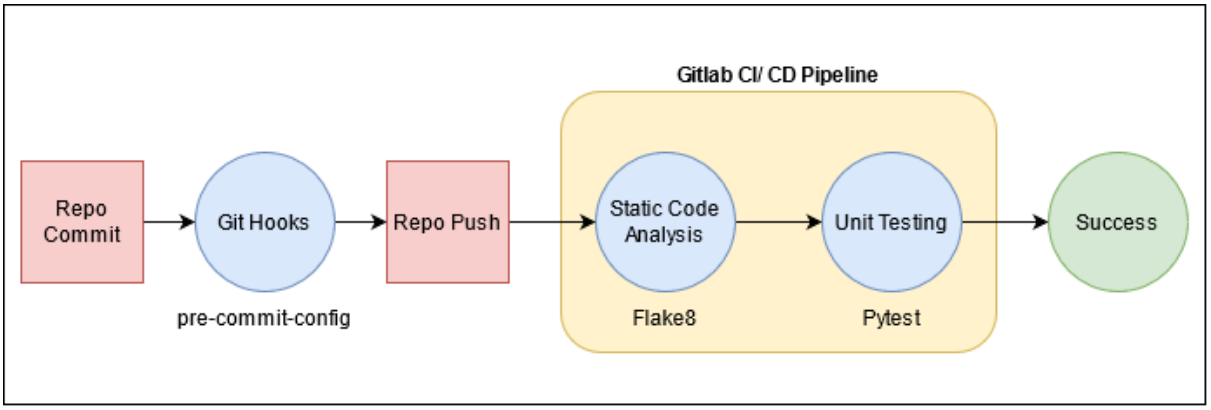

Figure 2: Unit Testing Flow

Testing Blacklist Functionality

These tests are designed to testing returning URLs either from input of the user when adding URLs to the blacklist in the preferences menu, using this blacklist to filter these URLs from sites to be scraped and using the domain name and suffix as labels when displaying the graphs in the application GUI.

Location: src/tests/test_blacklist.py

Test	Purpose	Result
test_base	Testing the ‘base’ functionality to return URLs to their base form of domain followed by the suffix.	PASS
test_blacklist_checker	Testing the blacklist logic, blacklisted URLs are provided along with a browsing URL list, no blacklisted URLs should be present in the filtered list.	PASS
test_base_with_blacklist	Testing integration of the above base URL functionality with the blacklisting filtering feature. Filtering URLs reduced to their base form from a list of URLs	PASS
test_different_googles	Using google as an example to test the overall blacklist and base URL functionality and logic. Google URLs often feature prefixes and have complex domains.	PASS

Figure 3: tests for the blacklist function

Testing Browsers

These tests were implemented to test that the SQL commands required to access the SQLite databases for different browsers worked and that the paths to browsers could be accessed. File structures were replicated with testing databases for this set of tests.

Location: src/tests/test_browsers.py

Test	Purpose	Result
test_firefox_linux	Testing firefox browsing history extraction on linux platforms	PASS
test_chrome_linux	Testing chrome browsing history extraction on linux platforms	PASS
test_safari_mac	Testing safari browsing history extraction on apple mac	PASS
test_edge_windows	Testing edge browsing history extraction on the windows platform	PASS
test_opera_windows	Testing opera browsing history extraction on windows	PASS
test_brave_windows	Testing brave browsing history extraction on windows platforms	PASS
test_firefox_windows	Testing firefox browsing history extraction on windows platforms	PASS

Figure 4: tests for the browsers

Testing Browsing Period Filtering

These tests were defined to test the browsing period filter feature of eMot. Output messages and results were checked to ensure that passing a combination of browsing period filter and browser choice returned the correct result.

Location: src/tests/test_history.py

Test	Purpose	Result
test_capitalise	Testing that browser tites were correctly formatted when provided as input to the system	PASS
test_browser_installed	Testing the browser and time period on a browser that was not installed locally on the test machine.	PASS
test_filter_error	Providing an incorrect browsing period option and checking system output format.	PASS
test_filter_spelling	Testing incorrectly supplied browsing filter option.	PASS
test_browser_spelling	Providing incorrect options for the selected browser input.	PASS

Figure 5: tests for getting the history

Testing Configuration File Integration

This test set is defined to test the access and modification of the JSON file used by eMot for filtering of blacklisted URLs and HTML tags.

Location: src/tests/test_json_sets.py

Test	Purpose	Result
test_get_first_url	Test designed to check URL list structure when accessing in a specific order.	PASS
test_get_first_tag	Test designed to check tag list structure by accessing in a specified order.	PASS
test_add_url	Test designed to add a URL to the URL list and check for its presence as the last URL added.	PASS
test_duplicate_url	Test designed to test the behaviour expected when adding a URL to the list when it is already present	PASS
test_add_base_url	Testing adding a URL using the base function to change the URL to domain and suffix only.	PASS
test_add_tag	Testing correctness of adding a HTML tag to be ignored when scraping.	PASS
test_add_duplicate_tag	Testing adding a duplicate tag already present and expected behaviour.	PASS
test_remove_tag	Testing the removal of a tag from the tag list.	PASS
test_absent_tag	Test designed to check for behaviour when trying to delete a HTML tag not present in the tag set.	PASS

Figure 6: tests for checking json manipulation

Test Preprocess

Tests designed to check the functions used when cleaning datasets for training classifiers as well as preprocessing functions used to clean scraped text and remove noise.

Location: src/tests/test_pre_process.py

Test	Purpose	Result
test_pre_process	Testing the preprocess function used to remove noise such as punctuation from scraped text.	PASS
test_remove_url	Testing the remove or hyperlinks, used in datasets and scraped text cleaning.	PASS
test_spell_check	Testing the spell checking behaviour of the spellcheck library.	PASS
test_clean_and_stem	Testing text noise removal and stemming functionality.	PASS
test_clean_text	Testing the integration of multiple individual text cleaning functions seen here.	PASS
test_wrong_clean	Testing the behaviour of functions when providing an obscure sentence structure.	PASS

Figure 7: tests for checking text processing

Test Sentiment Model

This test set is designed to check the classifier models for functionality and for expected labelling to manually provided sentences.

Location: src/tests/test_sentiment_model.py

Test	Purpose	Result
test_emotion	Checking if the predicted emotion matches our emotional understanding of the supplied text.	PASS
test_emotion2	Another example of manually testing the expected output classification label.	PASS
test_happiness_sentiment	Explicitly providing sentences expressing happiness to the classifier to check for correctness.	PASS
test_sadness_sentiment	Testing classifier labelling on sentences exhibiting sadness	PASS

Figure 8: tests for asserting sentiment expectations

Classifier Tests

Classifier Accuracy

The model training accuracy was tested by splitting the dataset used. A split of 80% training data and 20% testing data was used. Resulting classifier accuracy was 70% with an F1 score of 0.69 [1]. To check the balance of emotion scoring accuracy across the six emotions a confusion matrix was printed after each train [2]. This allowed us to view if some emotions were being confused for other emotions.

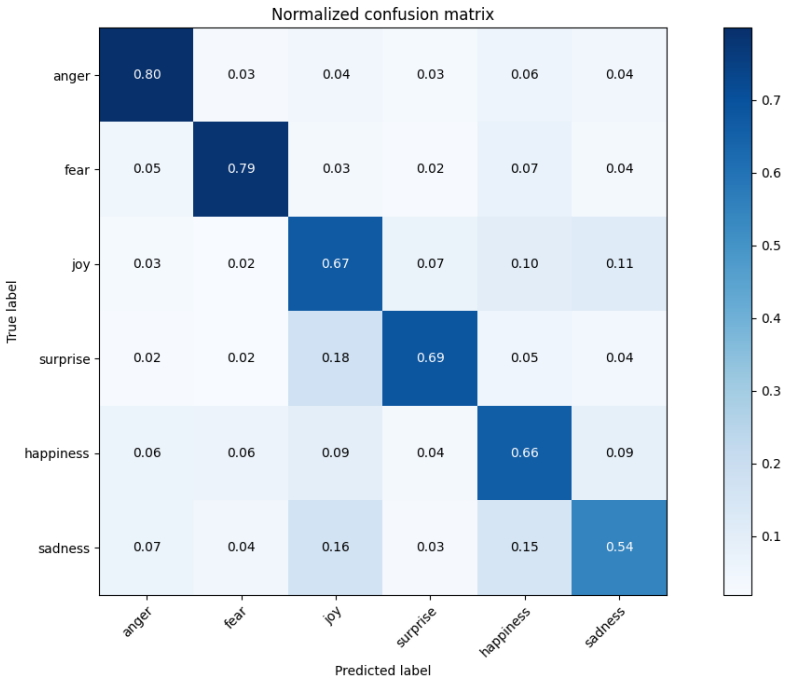

Figure 9: An example Confusion Matrix

Classifier Profiling

To get an idea of the performance of the classifier we ran some timed tests of execution when classifying some text. We provided differing lengths of sentences to see the effect they would have. Sentences ranged from 6 words to 12 words in length. We could see that classifying text may prove to be a bottleneck to program performance.

Test	Purpose	Result
profile_classifier	Profile the performance of the classifier on longer sentences, twelve words maximum.	0.0102 seconds
profile_classifier_2	Profile the classifier performance on short sentences, six words maximum.	0.006 seconds

Figure 10: Classifier Profiling results

Manual GUI Tests

These manual tests ensured that GUI button functionality was correct when using the application. The tests checked the button functionality using the file tab on the start screen of the emot application.

Test	Function	Result
Preference page tests	Testing the integration of PyQt when taking text input from the frontend and adding this to the blacklist tags file.	PASS
About page check test	testing whether the eMot about page can be accessed and that the manual hyperlink can be clicked successfully.	PASS
New application instance test	Testing whether creating a new instance of the application performed as expected.	PASS

Figure 11: manual tests for file tab functionality

These tests checked if items selected from the browser and time period values selected from the drop down menu were passed to the backend functionality correctly.

Test	Purpose	Result
Test browser input	Check if the item selected from the browser dropdown menu matched what was passed to the backend.	PASS
Test browsing period input.	Check if the item selected from the browsing period dropdown menu matched what was passed to the backend.	PASS

Figure 12: manual tests of message passing to backend

Testing tests analysed the graphical representation of the browsing data for correctness and button functionality.

Test	Result
Testing that correct data was displayed in each graph in the correct order.	Pass
Testing if the most positive and negative site were clickable hyperlinks	PASS
Testing if the graphs displayed scaled to different window sizes and ratios.	PASS
Testing if the next and previous page buttons changed to the correct page based on the current selected page.	PASS

Figure 13: manual tests of graphical representation of browsing data

User Testing

Each user had the option of installing the system on their own machine or trying the application from our PC's. The system was not compatible with some older versions of Mac and Windows and so all users decided to interact with a version of eMot installed on our machines. These users were presented with the application over Zoom remote desktop.

After the user interacted with the system we asked them to fill out a Google Form to collect feedback that they had. User Testing proved to be pivotal for the UI overall look and feel.

The most valuable feedback collected from the user testing was:

- Change graph colours, sometimes they are difficult to distinguish from one another.
- Label graphs better because they can be confusing what is being represented if the user manual is not closely inspected.
- The graphs were hard to understand because the title text was very small.
- There is too much information and not enough text explaining the graphs.
- Add more status tips to the pages when hovering over objects.
- The About page should link to the user manual instead of the repo because it is not publicly accessible.

The main elements of the user feedback were considered and the most viable options included in the project. User feedback allowed us to restructure the graphical layout of the application, spacing the visual aspects of metrics reporting with some supporting textual examples.

The results of the user testing feedback are available here.

References

[1] “sklearn.metrics.f1_score — scikit-learn 0.24.2 documentation.” https://scikit-learn.org/stable/modules/generated/sklearn.metrics.f1_score.html (accessed May 02, 2021).

[2] “sklearn.metrics.confusion_matrix — scikit-learn 0.24.2 documentation.” https://scikit-learn.org/stable/modules/generated/sklearn.metrics.confusion_matrix.html (accessed May 02, 2021).