Testing

Testing Strategy

While developing our application we added unit tests to newly developed functions and ran these tests every time we refactored or added functionality. Once the pages of the web application were developed we made integration tests that checked that the url, views and pages worked. These tests ran continuously as new features were implemented to the project. Unit and integration tests were made using django's "test" library, while performance tests were made using the developer console on Chromium-based and Firefox browsers. This allowed us to see the timings for various requests and identify areas where we could speed up performance.

Unit testing

Unit tests were developed for several files, including "addition.py", "decoder.py", "logisticreg.py", "searcher.py" and "selector.py". Unit tests contained both positive and negative unit tests. This means that the functions were verified to handle both the expected and unexpected inputs.

The tests were developed as the functions were created. As our first steps were to open and parse JSONL files, develop the backend functions to add tweets to the database and select them to be used in the frontend of the application, these were the first features to be tested. These tests belonged in the files "decoder.py", "addition.py" and "selector.py". The tests for "decoder.py" used a test JSON file to verify correct functionality. To test "addition.py", which contains the functions to add tweets to the database, test tweets were created just for testing and functions using these functions were mocked.

The results from running the tests "test_addition.py":

```
----------------------------------------------------------------------
Ran 3 tests in 0.015s

OK
```

To test "selector.py", which contains functions to return the polarity for days, months and key_dates, mocked functions were used to ensure only the target function was being tested.

The results from running the tests "test_selector.py":

```
----------------------------------------------------------------------
Ran 11 tests in 0.038s

OK
```

The next set of tests was for the frontend of the application, located in "searcher.py", which contained functions to select and render HTML files from the "templates" folder to the website. The functions to open and read files were mocked and the tests were ensured to work.

The results from running “test_searcher.py” are:

```
----------------------------------------------------------------------
Ran 7 tests in 0.035s

OK
```

The last unit tests were made for “logisticreg.py”, which contained functions for creating word clouds, removing stop words, tokenizing input, producing confusion matrices and training a linear regression model. These tests were created to ensure the functions were developed correctly as they were added and refined.

The results from running “test_lr_model.py” are:

```
----------------------------------------------------------------------
Ran 4 tests in 1.532s

OK
```

The results from running all tests are:

```
----------------------------------------------------------------------
Ran 28 tests in 1.519s

OK
```

Integration testing

Integration tests were made for all functions in “views.py”. These functions define the appropriate responses for handling HTTP requests. When a user requests a URL, the corresponding view is called, which interacts with the model and template, finally rendering a template.

The tests for these views were developed in the "test_views.py" file and were continuously run after changes were made to the functions. The tests mocked the dependent functions and ensured that the correct status code was returned, the correct HTML file was used in the response and the response context was correct.

The results from running the integration tests are shown below:

```
----------------------------------------------------------------------
Ran 5 tests in 4.697s

OK
```

Results from running all unit tests and integration tests:

```
----------------------------------------------------------------------
Ran 33 tests in 4.513s

OK
```

Performance testing

For performance testing we used the developer console on Chromium-based and Firefox browsers. This allowed us to see the timings for various requests and identify areas where we could speed up performance, for example by precalculating graphs and functions where they take up too much load time. Performance testing also allowed us to catch and prevent any memory leaks that could result from repeatedly drawing matplotlib plots. The performance testing results are given below.

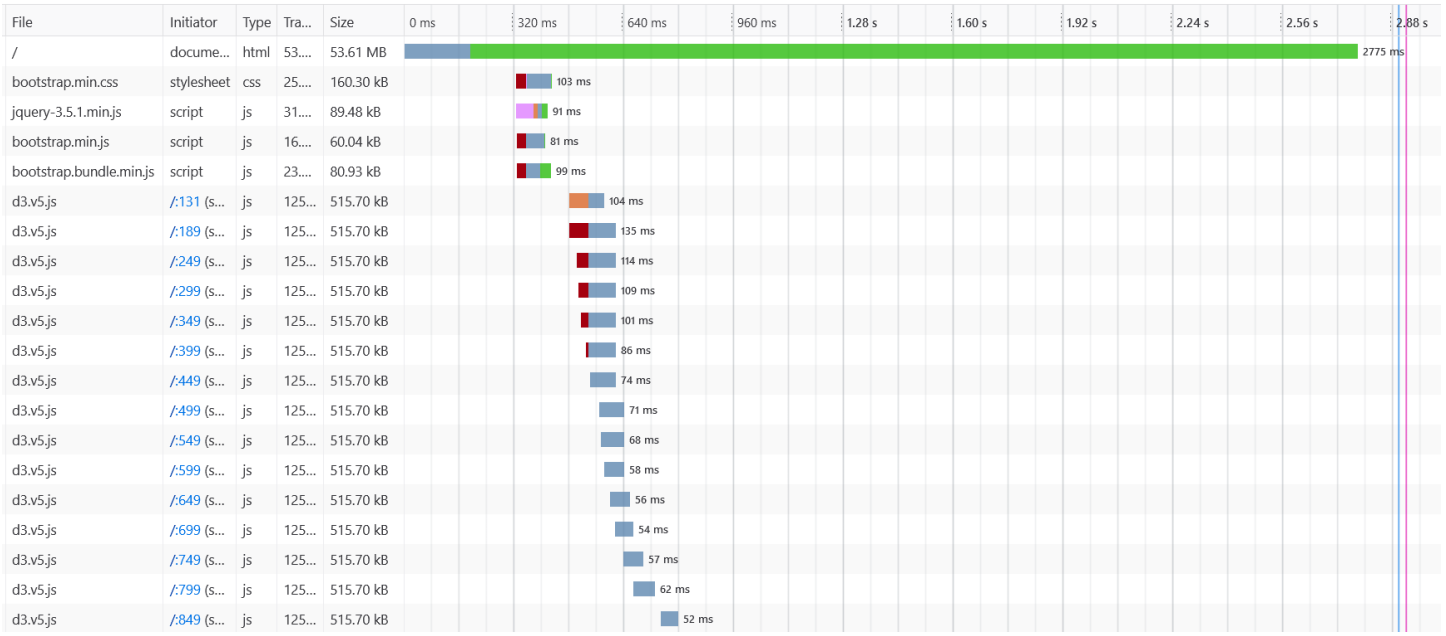
Local Testing (Cache Disabled)

Home page (Using real-time heatmaps):

Total requests:	54
Load time:	16.38s
Slow Requests:	3

Home page (Using pre-calculated heatmaps):

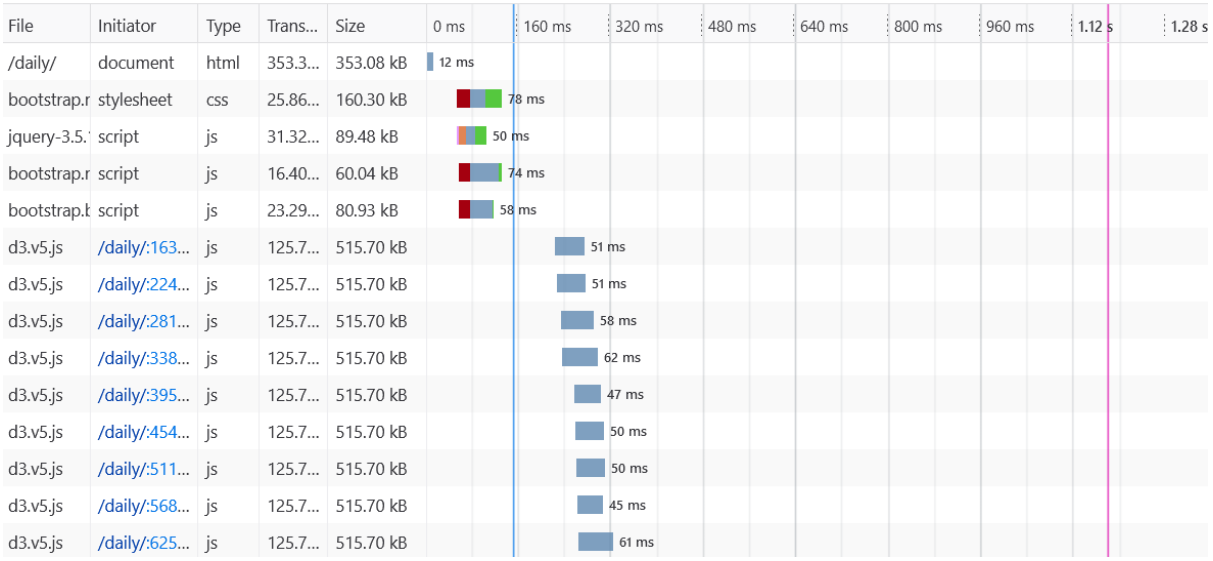
Total requests: 42
Load time: 3.17s
Slow Requests: 0



We found that using pre-calculated tweet frequency heatmaps improved performance significantly. This can be attributed to the large size of our dataset, numbering around ~400,000 tweets, and the processing required to get the data into a format suitable for heatmaps.

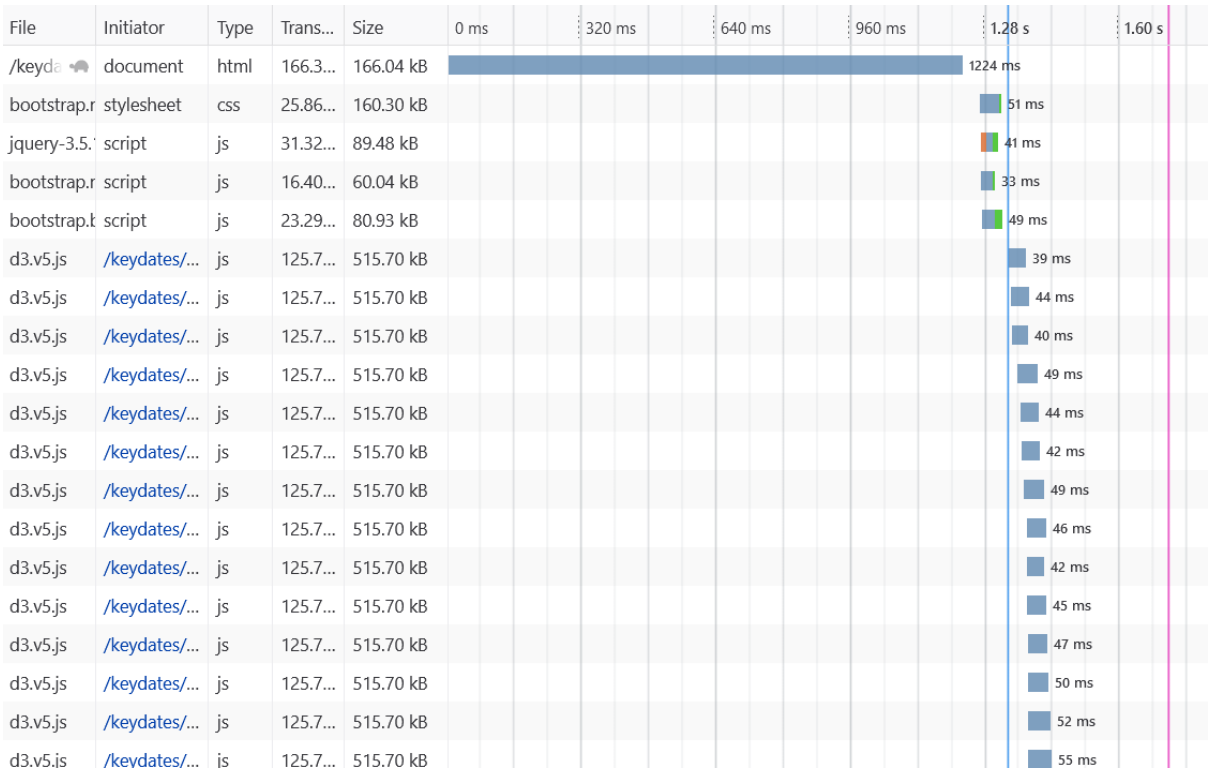
Daily page:

Total requests: 72
Load time: 1.16s
Slow Requests: 0



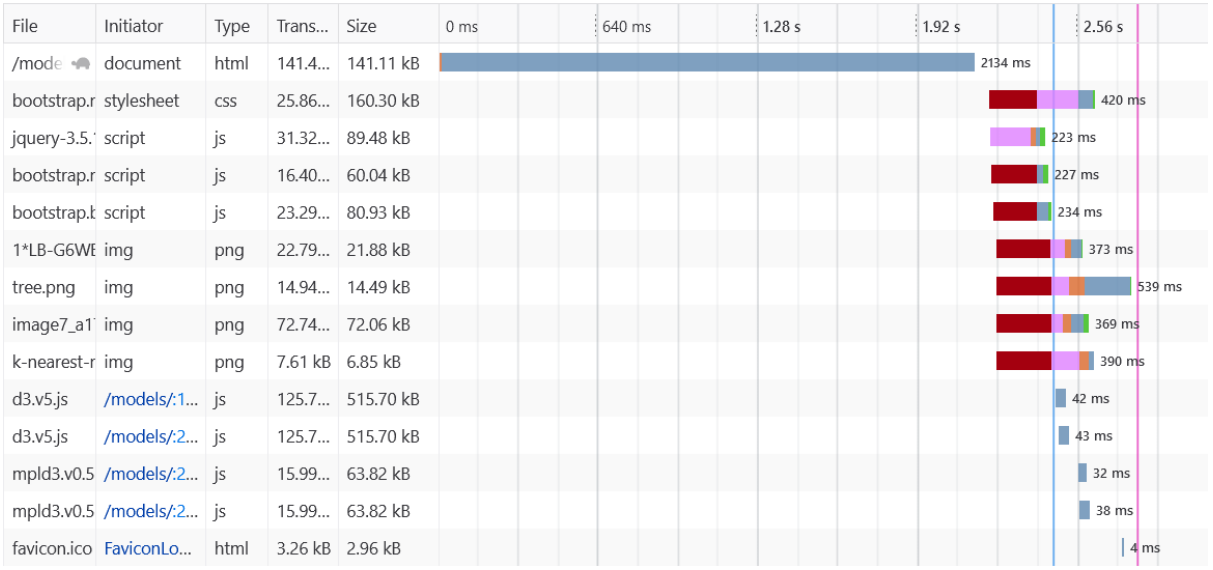
Key Dates Page:

Total requests: 34
Load time: 1.7s
Slow Requests: 1



Models Page:

Total requests: 14
Load time: 2.77s
Slow Requests: 1



About Page:

Total requests: 6
Load time: 236ms
Slow Requests: 0