1 Experiments and Testing

The designed and implemented system was practically applied to the semantic classification of the tweets regarding the discussion of "Britain Leaving the European Union".

1.1 Tweet Collection Experiment

Previous system's as stated in the related work used a set group of seed hashtags for which they collected tweets for and used as distant labels in their classification models. Instead of manually choosing the seed hashtags like previous Systems the experimentation choice was taken to extrapolate a seed hashtag set to use for collection and distant labelling. Initially, tweets containing either #Brexit, #VoteLeave and #VoteRemain were collected. After collecting tweets from the day of referendum (23/06/2016) up until the initiation of the Brexit negotiations (18/06/2017), the Jacquard's Distance Function (see Figure 1.1) was applied on each of the hashtags found within the tweets to compare the similarity of each of the tweets containing the hashtags discovered and the tweets containing seed hashtags. The result was **similar_hashtags.csv**; an ordered matrix of discovered hashtags that were similar to one or more of the seed hashtags. From the resulting similar matrix, a bigger seed hashtag set was chosen (see Figure 1.2) whilst manually ignoring anomalous hashtags which were outliers and not on the discussion topic.

$$\forall x.\ \forall y.\ d_J(x,y) = 1 - J(x,y) = 1 - \frac{|x \cap y|}{|x \cup y|} = 1 - \frac{|x \cap y|}{|x| + |y| - |x \cap y|}$$

$$where\ x = Set(tweets\ containing\ a\ unique\ discovered\ hashtag),$$

$$y = Set(tweets\ containing\ a\ seed\ hashtag)$$

$$and\ d_J(x,y) = Jacquard\ Distance\ Function$$

Figure 1.1: Jacquard Distance Formula Function.

#BREXIT #ARTICLE50 #BREXITDEBATE #BREXITMEANSBREXIT
#EUREFERENDUM #EUREF #VOTELEAVE #BREXITEER #IVOTEDLEAVE
#LEAVEEU #TAKEBACKCONTROL #VOTEOUT #STRONGERIN #REMAIN
#REMOANER #REMAININEU #VOTEREMAIN #STRONGERTOGETHER

Figure 1.2: Generated Seed Hashtag Set.

1.2 Testing Methodologies

The goal of testing within the System was to develop an effective test suite to verify and automate the testing of System behaviour. Tests were written for new functionality as the System functionality was developed, and thus the test methodology of Regression Testing was employed to ‘ provide a certain confidence that no new errors are introduced into previously tested code’ (**?**) when new changes to the system were implemented. The act of regression testing consists of running the entire suite of tests against new versions of the System to ensure expected System behaviour is upheld and was achieved through running of the test suite manual during local development of the System and automatically during the deployment of the System via a test step within the deployment pipeline (see Continuous Integration).

1.3 Testing Strategy

The System was tested using a mix of functional and non-functional testing, the types of testing carried out and the area for which they test is summarized in table /ref

Func. / Non-Func.	Testing Type	Scope of Test
Functional	Unit	Python Modules of Code
Functional	Integration	Interaction between Python Modules
Functional	Validation	Correctness of Designed Classification Models
Non-Functional	Load	Impact of Load on Usability of the website and operation execution time
Both	Acceptance	Ensure all requirements are met.

Table 1.1: Testing Plan

Unit Testing

The System’s test suite contains 44 unit tests that each test the functionality of python modules of code in isolation which can be found in each application’s file structure of `tests/unit/`.

Integration Testing

The System’s test suite contains 28 integration tests that each test the interactions between python modules and the database itself which can be found in each application’s file structure of `tests/integration/`.

Model Validation Testing

The System as part of training the models conducted Holdout validation of an 80:20 split between training and validation data. The training and validation data separation was picked at random on an execution of a single Holdout run. As part of validating the models ten repeated Holdout runs were conducted on ten different versions of the model which resulted in the following average statistics across the models.

Naive Brexit Model

Class	Precision	Recall	F1 Score	Training Samples
Leave	0.71	0.69	0.70	1382
Remain	0.70	0.71	0.70	1383
Avg/Total	0.705	0.70	0.70	2765

Table 1.2: Naive Brexit Model Results

Sentiment Model

Class	Precision	Recall	F1 Score	Training Samples
Positive	0.68	0.53	0.60	371
Negative	0.62	0.75	0.68	371
Avg/Total	0.65	0.64	0.64	742

Table 1.3: Sentiment Model Results

Load Testing

To ensure System satisfied **NFREQ-2**, the system was load tested to mock what common daily usage of the website may entail and inspect how load effects the usability of the website. Using the python load test package Locust, a unique page of each type was specified within a set of pages to load test (although not a conclusive set, it gives a good general idea to usage of every page). The load-test environment was spun up with 100 unique users and reached an average of 35 requests per second. The load-test was continued until no huge change was seen within response times. The results of the load test concluded that **the site satisfies NFREQ-2 by consistently responding at around an average of 2.2 seconds**. For a further breakdown of the response times see Figure 1.3

Type	Name	# requests	# fails	Median (ms)	Average (ms)	Min (ms)	Max (ms)	Content Size	# reqs/sec
GET	/	622	0	2200	2182	31	8500	10454	4.4
GET	/about	527	0	2000	2009	26	5997	6208	4.7
GET	/analysis	557	0	2200	2192	29	5934	5933	4.9
GET	/analysis/brexit_stance/day/?time_period=28/02/2017	574	0	2000	2154	41	6198	10852	3.7
GET	/analysis/brexit_stance/month/?time_period=03/2017	573	0	2300	2259	40	6204	24373	3.8
GET	/analysis/brexit_stance/week/?time_period=27/02/2017-05/03/2017	593	0	2300	2242	56	19037	26003	5.8
GET	/analysis/brexit_stance/year/?time_period=2017	589	0	2300	2285	47	7644	30966	4.6
GET	/data	595	0	2300	2175	51	5412	34334	3.1
	Total	**4630**	**0**	**2200**	**2190**	**26**	**19037**	**18868**	**35**

Figure 1.3: Locust view of Load Testing.

Acceptance Testing

To test against the functional and non-functional requirements identified within the Software Requirements Specification, a mix of black box and white box testing was used to comprehensively test against the set of all requirements. The result of such testing found that all requirements as identified within the system had been met.

– See Table 1.4 for how the System met the requirements.
– See Table 1.5 for how the System met the non-functional requirements.

Requirement Set	REQ	Implementation
1. Tweet Collection	1.1	The System scrapers allow for the specification of hashtags for which to collect tweets for.
	1.2	The System uses tweet selection criteria in both historic scraper and realtime scraper
	1.3	The System permanently stores the tweets it collects using the `Tweet Model` as a means of saving.
2. Tweet Tokenization	2.1	The System provides a tokenization interface for manual tweet tokenization.
	2.2	The System tokenization interface accepts tweet selection criteria
	2.3	The System can tokenize a plaintext tweet into a classifiable string of tokens
	2.4	The System permanently stores the tweets it tokenizes using the `TokenizedTweet Model` as a means of saving.
3. Tweet Classification	3.1	The System provides a classification interface for manual tweet classification.
	3.2	The System classification interface accepts tweet selection criteria
	3.3	The System contains a `Naive Brexit Model` and `Brexit Model` which classifies tweets on whether they are in favour (leave) or against (remain) Britain Leaving the European union
	3.4	The System contains a `Sentiment Model` wgucg classifies tweets on whether they are positive or negative towards Britain Leaving the European union.
	3.5	The System permanently stores the tweets it tokenizes using the `TokenizedTweet Model` as a means of saving.
	3.6	The System permanently stores the tweets it classifies using the `ClassifiedTweet Model` as a means of saving.
4. Visualization	4.1	The website displays all tweets the system has collected on page `data.html`.
	4.2	The System provides `analysis.html` which allows the filtering of analysis by model
	4.3	The System provides a time switcher on `classifier-analysis.html` for which analysis can be filtered by day/week/month/year
	4.4	The System displays all classifications but only a subset of top classified tweets.
5. Automation	5.1	The System conducts tweet collection autonomously using the `real-time` scraper implementation.
	5.2	The System conducts tweet tokenization autonomously using asynchronous celery task `receive tweet`
	5.3	The System conducts tweet classification autonomously using cron job `brexit/brexit/scripts/cron.sh`
	5.4	The System conducts tweet visualisation autonomously through dynamic `template pages`

Table 1.4: Requirement Implementation

NF-REQ	Implementation
1	The System operates in real-time and background real-time operations occur within five minutes.
2	The System on average responds to HTTP requests in less than 3 seconds and has been tested with load testing.
3	The System works across common operating systems.
4	The System works across common browsers.
5	The System uses JSON configuration file that allow specifying models.
6	The System uses Django's data models which are extensible through South Migrations.
7	The System uses supervisor to ensure it is reliably up and operating.

Table 1.5: Non-Functional Requirement Implementation

Continuous Integration

For the automated checking of the formally written unit and integration tests within the system the continuous integration environment *Jenkins* was used. Jenkins as previously mentioned within *Section* **??** allows for the specification of a delivery pipeline, for ensuring System correctness during deployments a test step was put into the pipeline. Upon executing the test step of the delivery pipeline, each of the tests contained in the System is run against the new System to ensure the previous behaviour of the System is upheld when new changes have been implemented. If the tests fail within the execution of the test step, Jenkins records the build as a fail, and the deployment is abandoned allowing for the problematic changes to be caught early. Continuous integration ensures the system can be tested in a clean environment (created by the build step) on the Jenkins server thus providing confidence in the functionality of the System as the Jenkins environment is reflective of that of the final destination of the remote local production server. See Figure 1.4 for the Jenkins view of continuous integration testing of previous deployments through a test result trend chart.

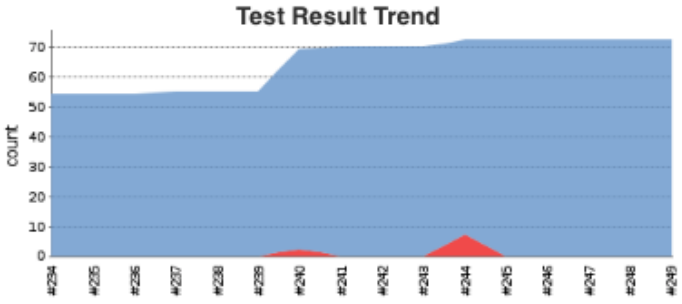

Figure 1.4: Jenkins view of Continuous Integration testing of deployments.