1 Implementation

1.1 Software Stack Selection

1.1.1 Programming Language

The System as analyzed in the specification requires a number of common computational tasks including web-scraping, natural language processing, machine learning, database interfacing and task execution handlers. To avoid reinventing the wheel within the scope of the system, the design decision of utilizing libraries where possible was taken. Libraries for the above-mentioned operations exist across popular programming languages however through further analysis Python became the obvious candidate. Python as a language promotes the integration of existing Python modules and 'has an active supporting community of contributors and users that also make their software available for other Python developers to use under open source license terms' [1].

1.1.2 Framework

The componentized system as highlighted in Figure **??** consists entirely of back-end components except for the singular visualizer front-end component. Due to the API nature of the back-end interfacing with the front-end, there was room to have a complete separation between the back-end and the front-end where a Python back-end would power a front-end framework that is completely detached from the business logic of the back-end system. However, the bulk of the requirements laid in the back-end and thus having a separate framework for the front-end could be argued to be overcomplicating the implementation of what is a well-structured System architecture. The design rationale was therefore taken to implement the back-end and front-end in a singular framework whilst upholding the design of having the business logic and API in the back-end allowing future extensibility of pulling the front-end into a separate framework at a later date if the System required it.

Having decided on a singular framework for the system, the approach was taken to research and compare the top three python frameworks that would offer full stack capabilities along with a number of identified desirable quantities needed in the framework. Using the comparisons conducted in Table 1.1 and further research the framework decided upon was Django for it's out of the box capabilities and separate application support being most appropriate for the proposed solution (batteries included).

[1] Python Modules accessible via https://docs.python.org/3/installing

Full Stack Python Web Framework Comparison					
Name	Python Version Required	Separate Application Support	Templating	Batteries Included	Github Stars
Django	3.X	Yes	Yes	Yes	32,770
Flask	≥ 3.3	No	Yes	No	34,207
Pyramid	≥ 3.4	No	Yes	No	2,700

Table 1.1: Python Framework Comparison **as of publishing date**

1.2 System Architecture

Through the implementation of the Python full-stack framework "Django", the system is encapsulated in a set of Python modules separated into "applications" that represent the high-level components of the system. Each application is registered within the system and contains a number of Python classes which are responsible for providing the component responsibilities that the application represents. The System also contains utility modules which are used across all the applications and therefore are available at the top level of the system. Along with applications, there are multiple configuration files and scripts that enable the remote execution of the system.

1.2.1 Applications

The applications provided by the system and the component that they represent are as follows:

- **classifiers**: representative of the classifier framework component.
- **core**: representative of the visualizer component.
- **tokenizer**: representative of the preprocessing framework component.
- **scraper**: representative of the scraper component.

Each application contains a series of classes that provide the functionality required for the application and some Django specific classes for configuration (non-capitalized). If the application interfaces with the database it will contain a "models.py" file which are Django representations of the database tables that the application owns. Each application also contains a "views.py" file which also forms part of the visualizer component as each applications "views.py" file provides API endpoints that populate the template pages when requested.

1.2.2 Dependency Injection

The system contains two methods of dependency injection to provide the system access to third-party libraries of established code, it utilizes a python virtual environment to inject python libraries into the system for use within the back-end components of the system and nodejs is used to inject javascript and CSS libraries into the system for using within the front-end visualizer component of the system. On starting the system, the dependencies are copied over to the global "static/" folder from which they are globally accessible from within the system. The configuration files for dependency injection are `requirements.txt` (back-end injection) and `package.json` (front-end injection) which both contain a list of dependencies and there versions utilized by the system.

N.B. Third party libraries that have been used without dependency injection have been clearly stated within the code using a header comment.

1.3 Database Implementation

The System contains two database implementations, the live production PostgreSQL database hosted via AWS and the auxiliary test SQLite database used for integration testing of the system. The implementation of database structure required the translation of each model in the entity relationship diagrams into Django Models such that the System could then utilize the built-in object-relational mapping (ORM) tool to easily access database records without the need for SQL. Django handles the database management of the system with changes to the models occurring through *South Migrations* which are then propagated to the database. The entities from the physical entity-relationship diagram were translated into Django Models with each attribute from the entity being a field of the representative Django Model. Through translation of the data models, a number of constraints were also implemented to enforce the data integrity within the database. For illustration purposes the translation of the Tweet entity can be seen in Figure 1.1

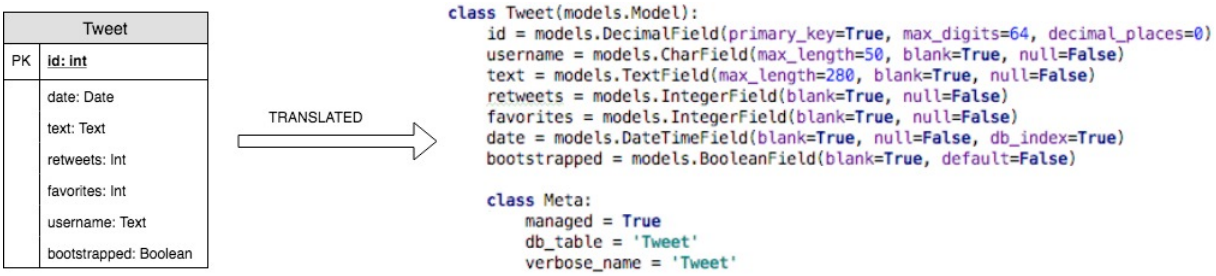


Figure 1.1: Data Model Translation.

1.4 System Features

The System features that were outlined in the proposed specification have their implementations listed below.

Tweet Collection

The scraper application hosts two interfaces with Twitter for the collection of tweets. Having two application product interfaces (that communicate with Twitter) for the collection of tweets allows for the collection of **historic** and **current (in real-time)** tweets.

Historic Scraper

The system integrates the existing third party tweet collector named "GetOldTweets-python" [2] that allows for the backdating of tweets older than two weeks. It does this by manually scraping the JSON of the twitter advanced search timeline using the requested dates, languages and hashtags as filter criteria. The historic interface was amended to work within the Django framework and optimized using the system's concurrency utilities to allow for thread pool execution of the extraction of tweets from the JSON as well as asynchronous saving of tweet objects to the database.

Real Time Scraper

The system integrates the existing third party tweet collector named "Tweepy" [3] that allows for the collection of tweets in real time i.e. as they are posted. Tweepy allows the system to communicate with twitters firehose (tweet access) provider "gnip" and request access to real time tweets for a specified language and hashtags.

Tweepy enables a connection between the client application and gnip to be established, then queries the streaming API with the filter criteria resulting in the delivery of tweets to the client application as they are posted and upon close of the connection attempts to retry connecting until a new connection is established and the collection restarts. The raw tweet JSON objects are placed in a Redis message broker to process the saving of tweets asynchronously via a celery application. The asynchronous processing is key to the operating of real-time collection as gnip delivers tweets from a fixed size queue which only holds tweets for a limited time before disposing of them. Celery (an asynchronous task executor) then processes the receive tweet task in which the raw tweet JSON is converted into a tweet model object and then saved to the database.

[2] GetOldTweets-python accessible via https://github.com/Jefferson-Henrique/GetOldTweets-python
[3] Tweepy accessible via http://www.tweepy.org/

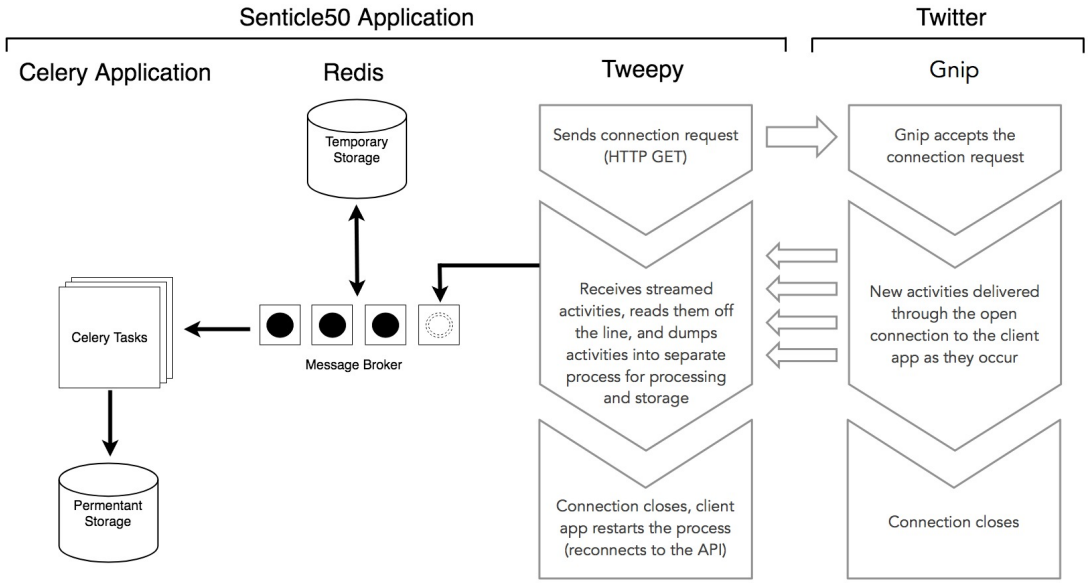


Figure 1.2: Real-Time Tweet Collection [4].

Tweet Tokenization

The tokenizer application implements the preprocessing step of tweet classification, it manipulates the raw text of tweet objects and converts tweets into a "Tokenized Form".

Each Tweet is converted into a string of "Tokens" through a series of tokenization steps that utilize regular expressions and the natural language toolkit to accomplish this. A tweet's raw text will therefore go through the following steps in order

1. All text is lowered in case.
2. Twitter Mentions are replaced e.g. @UserMention.
3. Hashtags are removed e.g. #Example.
4. Twitter "RT" symbol for retweet is removed.
5. Extra whitespace is removed
6. URL's are removed e.g. http://url.com
7. Stop words are removed e.g. his/her.

[4]Modified Image accessible via http://support.gnip.com/articles/consuming-streaming-data.html

8. Punctuation is removed e.g. ! , / ' ( )
9. Remaining text is split into an array of tokens (words)
10. Words are stemmed to reduce dimensionality of feature set e.g. running -> run
11. Final tokens are joined back together as a token string.

Tweet Classification

The classifiers application implements the classification framework component of the system. The set of implemented classifier classes contains the `CombinationClassifier`, `KerasClassifier`, `StringContainsClassifier` and `StringEndClassifier`. Each of the the implemented classifiers classifies tweets against a model which is passed in as an attribute when instantiating the classifiers as mentioned in the Design. The classifiers application exposes two main services that can be interacted with, the `ClassifierService` for classifying tweets on a one tweet per classification basis and the `BatchClassifierService` for classifying larger groups of tweets on a batch basis. The classifiers application also provides two registry classes that implement the registry pattern discussed in the design; the `ClassifierRegistry` and `RealTimeClassifierRegistry`.

Model Architecture

The models that utilize a neural network i.e. the `Naive Brexit Model` and `Sentiment Model` have been created using the Neural Network library Keras. Each of the models convert the tokenized tweets into a padded vector of 140 length (that can represent the new max length of 280 characters within tweets). Each entry in the vector therefore represents the presence of a word using the index of the word from the tokenized feature set of the top five thousand words found in the training set of the model. Utilizing this structure in the model therefore takes the context (order) of the words as well as the presence of the words for learning patterns of order unique to the classes through the convolutional layers defined within the network. See the experiments application `management/commands/` folder for the model neural network structures.

Classifier Architecture

The architecture of classifiers has each of the classifiers implement the interface of a Classifier superclass. The set of implemented classifiers within the classifiers application are all subclasses of the abstract base class `Classifier`. Through the subclass relationship they are required to implement the abstract methods defined within "Classifier" and thus the interface relationship is enforced (See Listing .

```
from abc import ABC, abstractmethod

class Classifier(ABC):

    @abstractmethod
    def __init__(self, model_to_load):
        pass

    @abstractmethod
    def classify(self, text, model_parameters):
        pass

    @abstractmethod
    def classify_batch(self, texts, model_parameters):
        pass

    @abstractmethod
    def get_classifier_name(self):
        pass

@classifier
class KerasClassifier(Classifier):

    def __init__(self, model_to_load):
        ...

    def classify(self, text, model_parameters):
        ...

    def classify_batch(self, texts, model_parameters):
        ...

    def get_classifier_name(self):
        ...
```

Listing 1.1: An example of Classifier interface utilization.

Classifier Registries

The aforementioned classifier registries are populated by the scripts `RegisterClassifiers` and `RegisterRealTimeClassifiers` located in the scripts folder within the system wide configuration folder of `brexit/brexit/`. Each registry is an instantiation of a `Singleton` base class thus providing system wide access to the available classifiers after being populated on startup of the system.

```
class ClassifierRegistry(Singleton):

    def _init(self):
        self.classifiers = {}

    def get_registered_classifiers(self):
        return self.classifiers

    def add(self, classifier_name, classifier):
        if classifier_name not in self.classifiers:
            self.classifiers[classifier_name] = classifier

    def get(self, classifier_name):
        if classifier_name in self.classifiers:
            return self.classifiers[classifier_name]

        raise ValueError('Classifier not found in classifier registry')
```

Listing 1.2: ClassifierRegistry.py class implementation

The ClassifierRegistry provided by the classifiers application allows for a service to be generic by dynamically pulling the class implementation of the configured classifier and instantiating it.

```
classifier = ClassifierRegistry().get(config['classifier_name'])
```

Listing 1.3: Usage of ClassifierRegistry

The `RegisterClassifiers` script is executed first and populates the `ClassifierRegistry` with each of the available classifiers within the system that end in "Classifier". For a classifier to be registered within the `ClassifierRegistry` it is required to have the *classifier decorator* which allows for the classifier to be added to registry upon import.

```
def classifier(cls):
    classifier_registry = ClassifierRegistry()
    classifier_registry.add(cls.get_classifier_name(), cls)
    return cls

@classifier
class KerasClassifier(Classifier):
    ...
    @staticmethod
    def get_classifier_name():
        return "KerasClassifier"
```

Listing 1.4: Usage of "classifier" decorator

After the `ClassifierRegistry` is populated the `RegisterRealTimeClassifiers` script begins to populate the `RealTimeClassifierRegistry` with the json classification configuration files that contain the boolean flag "is_real _time" that is true. It is the configuration files that are contained in the `RealTimeClassifierRegistry` that are utilized in the real time classification of tweets.

Visualisation

The application core and the `views.py` files contained in the rest of the applications together make up the visualizer component of the system. The visualizer component as highlighted in Design is split into pages, with each page begin reachable by the navigation bar (visible at the top of the page) except for the classifier-analysis page which is accessed via buttons within the analysis page and thus is the only second depth page in the visual system. The visual component utilizes bootstrap for css conformity and cross browser/device layout support, jQuery for user interactions and D3 for the analysis pictorial visualisations displayed on the classifier-analysis page.

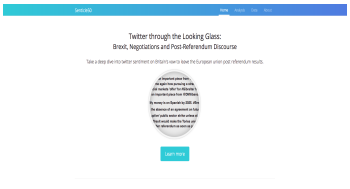
API

The system has a number of configured url paths which are registered in the system within `urls.py` files. These url paths are regular expressions that match the url of a given request and handle the execution of that url accordingly. If the url does not match any of the registered url paths within the system Django returns a page-not-found exception (HTTP 404) and renders the 404.html page found within the `templates/` folder. On the successful match of a url a `views.py` endpoint is called to render a template and JSON context that holds the data for the template to display.

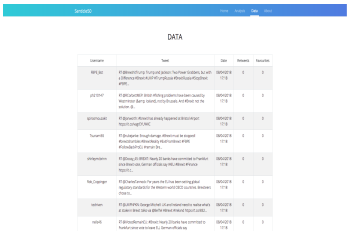
Template Pages

The System uses Django's built-in templating feature to populate pages where the information is displayed dynamically i.e. data displayed on the page is not static. The templating structure consists of a base template `index.html` that defines the overall layout of the website and includes the elements of the site which are viewable regardless of page e.g. navigation header and footer. The base template defines a body section which each child template page then extends with the content of that page. Each template page serves one url except the classifier-analysis page which serves four url's; one for each displayable time unit. Using a single template for the classifier-analysis page allows the rendering of four unique views of the page each of which has dynamic data populated through various permutations of the parameters classifier type and time period.

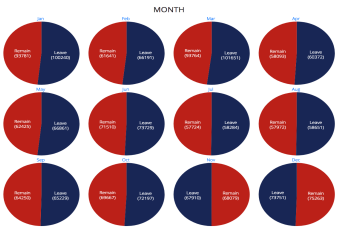
The final set of viewable implemented pages as outlined in the Design are as follows; `home.html`, `analysis.html`, `classifier-analysis.html`, `data.html` and `about.html`.



(a) home page



(b) data page



(c) analysis

Figure 1.3: System Website Screenshots.

Automation

The automation of the system is the flow between all of the components within the system. The System pipeline outlined in the Design has been implemented through a series of scripts that configured remotely on the server. The remote server uses a process control system named Supervisor to ensure all of the automated tasks are running. The system also uses a scheduled cron job to classify the new tweets it has received every five minutes.

The system automation is split into two tasks: the collection and tokenization of a tweet as well as the classification and visualisation of those tweets. As mentioned in the tweet collection system feature implementation, asynchronous tasks are used to convert the real-time collected tweets into tweet models. After conversion to a tweet model the tweet is also tokenized. The collection and tokenization of the tweets requires the running of the system and the Celery (asynchronous executor) application. To ensure both of these system's are running a Supervisor configuration was enabled for each of commands to run

the two systems required. It was found that the classification of tweets on a one tweet per classification basis was inefficient for real time classification as Keras required a lock when asked to predict a class given a sample, hence batch classification and visualisation was chosen to be a task of it's own. The batch classification and visualisation classifies and updates the composite visual objects required for visualisation on the tweets which haven't yet been classified today, this task runs every five minutes on a scheduled cron job.

1.5 Deployment

For the deployment of the system onto the remote virtual private server the continuous delivery environment *Jenkins* was used. Jenkins as a tool allows for the automation of tasks through the specification of a delivery pipeline. The System utilizes a `JenkinsFile` which specifies the continuous delivery pipeline consisting of a Build, Test and Deploy step (see Figure 1.3). Upon running the System's pipeline, Jenkins handles the pulling of the System's GitLab repository itself after which it is the Build and Deploy steps of the delivery pipeline which is required for deployment of the System via Jenkins. The Build step first installs a clean environment for the System to run in ensuring that all dependencies are injected, the static front-end files are generated and that the System can be built i.e. that it compiles. After a successful build the Deploy step handles across a secured ssh connection to the remote server; the removal of the old system, the copying of the new system files via the secure copy protocol and the restarting of the system through supervisor commands. Upon successful execution of the delivery pipeline the System is backup within seconds containing the new changes which had been committed to the System's GitLab repository prior to running the delivery pipeline.

N.B. If any steps within the delivery pipeline fail, the deployment of the system is aborted with Jenkins displaying the source of error thus preventing a faulty system from being deployed onto the live production server.

Figure 1.4: Jenkins view of the Continuous Delivery Pipeline.