eMot

Technical Guide

Michael Savage - 17313526

Gerard Slowey - 17349433

Supervisor: Jennifer Foster

Date Completed: 02/05/2021

Abstract

This project consists of a python GUI application which runs locally on the users machine. The program is designed to allow a user to analyse the emotional content of the text being read online by them over a defined time period. This content is analysed with metrics being graphically presented after analysis. This process is accomplished through the use of various project components such as a browser history extraction package, a web scraper, a javascript rendering engine, a combination of support vector machine and logistic regression classifiers and a python graphical user interface. The emotion classifiers are trained to detect the emotions ‘anger’, ‘fear’, ‘joy’, ‘surprise’, ‘happiness’ and ‘sadness’. The distribution of these emotions across the online text the user has read can be viewed within the application.

This project proved to be a challenge due to the difficulties associated with natural language processing and machine learning. It did prove to be a majorly beneficial undertaking however, introducing learning experiences in user interface design, logistic regression and support vector machine classification, application performance and scalability, database interaction, web scraping and data extraction and software testing principles. These learning experiences will prove very beneficial going forward when working in industry.

Motivation

The motivation for this project originated with both team members expressing an increasing interest in emotion classification during the summer prior to the current academic year. When deciding the project area we agreed that pursuing a concept centered around natural language processing and sentiment analysis would be beneficial and engaging for both parties.

A primary motivation that spurred the text material that would be analysed stemmed from the increased uptake of working from home as a result of the global Covid-19 pandemic. With people spending more time online, we thought it would be beneficial to be able to allow them to gain an insight into the text they were reading. This would allow a user to make informed decisions about their online reading habits.

Furthermore, from our research this project concept seemed like a novel idea and we liked the idea of taking on a challenge which had not been done in the mainstream before. Researching about the emotion classification of online text lead us to the classification of textual reviews on ecommerce sites such as amazon.co.uk and on restaurant review forums, an example of this can be seen here [1]. We therefore decided that the classification of a broader range of texts from different genres of websites would be a different approach to a classification problem.

Glossary

API	Application Programming Interfaces are a set of functions and procedures allowing the creation of applications that access the features or data application
CSV	A comma-separated values (CSV) file is a delimited text file that uses a comma to separate values. Each line of the file is a data record. Each record consists of one or more fields, separated by commas.
DOM	Document Object Model (DOM). JavaScript and other scripting languages determine the way the HTML in the received page is parsed into the DOM that represents the loaded web page.
GUI	Graphical User Interfaces are systems of interactive visual components for computer software.
HTTP	Hypertext Transfer Protocols allows users to fetch resources from servers, such as HTML documents. Clients send requests to servers and servers send responses back.
PYQT	A Python binding of the Qt C++ cross-platform framework. It is a GUI module.
SQL	Structured Query Languages allow you to access and manipulate databases.

Research

Project Viability

Our initial research involved determining whether a project of this type would be achievable and if it would be in scope for a final year software engineering project. As mentioned previously we had read several research papers and online articles which classified text from online sources [2], so this indicated to us that the emotion classification concept of this project would be feasible. We discussed the concept with our project supervisor and liked the feedback we received so decided to proceed.

Initially for a proof of concept we researched some off the shelf options of text classification in python, most notably the ‘NLTK’ and ‘TextBlob’ libraries. These provided a very quick and simple introduction to natural language processing. We used these libraries to build basic prototypes of a text classification system. These libraries seemed very useful but lacked customisation options and would limit our learning experiences in the project. We therefore decided to do research into implementing a sentiment analysis classifier and then eventually an emotion classifier.

Extracting Text Data

Our next challenge involved obtaining the text being viewed on screen by the user. The first option we considered involved designing a web browser extension that would be installed in the users browser [3]. This extension would access and record the text in the sites visited. The second option was to implement a direct text scraper run from the users machine.

We felt that the browser extension approach would limit the scope of browsers that we could develop these extensions for and would require a separate extension for different browsers. We therefore decided to go with a direct text scraper approach. This would allow for a more functional approach to extracting relevant text from sites visited and could be developed in a common language as part of the main project, helping to speed up development time.

URL Extraction

Since we had decided to implement a scraper as part of the project we needed a method to obtain a collection of browser history URLs. These would then be supplied to the ‘Splash’ to render which would then be scraped and segmented by beautiful soup. This is discussed in more detail in the next section.

Most modern web browsers store their browsing history files in a SQL database using SQLite. We started to interact with the browser SQLite databases using the tool SQLite Browser [4]. This GUI interface allowed for querying and manipulation of the browser database easily. So we decided that this would be the best method to access the users browsing history.

Site Rendering and Scraping

Having decided that a text scraper would be used for text extraction, we began searching for the most suitable text scraper that would suit our needs. The best options from our research were Selenium [5], Beautiful Soup [6] and Scrapy [7]. Initially we focused on Scrapy as it seemed to be the most feature rich, the fastest and the most versatile. However after testing and trial implementations it was difficult to configure and seemed to be designed to scrape text from whole web pages and child pages rather than being able to specify certain sections of a page to focus on.

Furthermore, during our initial prototype scraper implementation we encountered a scraping issue when visiting non-HTML dynamic based sites. With progressively more sites online switching to rendered DOM structures a traditional web scraper would not be sufficient to extract text from these. These sites require the users browser to render the information being displayed on screen. Running tests using a basic text scraper on these sorts of sites would return an empty HTML structure and therefore needed to be addressed. To overcome this we decided that a headless browser would be the most appropriate solution. From research we discovered ‘splash’ [8], splash is a lightweight stateless javascript rendering service based on headless chromium, which provides a HTTP API to access content on rendered sites. We decided to use the Beautiful Soup parser from the options above in combination with the splash renderer. This would provide an efficient solution to our problem, as early tests seemed promising.

System Customisation

For our system we wanted the user to be able to customise the system by providing some input. Some leading ideas we had were:

- Blacklisting sites to ignore while scraping
- Allowing the user to select a certain browser
- Specifying a specific time period

Storage of this information would be needed in order to preserve user preferences, so we researched suitable storage solutions that could be used. Our original idea involved using a configuration file [9], however we both agreed that this was too complex for our usage and that a simple file structure such as JSON or CSV would be more appropriate.

To allow the user to input their chosen browser and browser history period they wanted to analyse we initially started by taking text input from the user. While this solution worked the input needed to be changed to a standard format i.e. lower case. If there were any spelling errors this would not cause the system to fail, we therefore decided that some graphical menu which the user would choose from would be better, this was noted for future consideration.

Sentiment Analysis

To begin with, we started looking at sentiment analysis of text by working on a lexicon-based approach to sentiment analysis using Bing Liu's sentiment lexicon [10]. The scraped text was split into sentences and a set of negative and positive words were used to judge the sentiment of the text. We then explored ways to combine information sentiment lexicons into a sentiment score. We further investigated by looking at subjective versus objective datasets, and looked at sentiment analysis datasets that include the neutral label. By including a neutral class we transitioned from binomial to multinomial classification.

To enhance the analysis of the text being read we decided that our end goal for the project would involve emotion classification. We then needed to research our approach to classifying the emotional content of the text being read. This task would require the training of a machine learning classifier. Our project supervisor directed us to research the machine learning library scikit learn. A selection of the most common standard classification algorithms would be analysed to see which one was most suitable. We would take a supervised approach to machine learning to train classifiers such as Logistic Regression and Support Vector machines.

Emotion Datasets

When we focused our attention on training a classifier we researched possible datasets that could be used. Our supervisor informed us of a corpus of annotated datasets for emotion classification in text [11]. By analysing the associated paper [12] we saw that the dataset was a substantial size, and consisted of a mix of thirteen emotions. Additionally the corpus covered a wide range of labelled topics from social media posts to news media articles.

We needed to be sure that the data in the corpus would be representative of the text being read online. Since text online could contain any textual content and subject we thought it would be beneficial to select general datasets from the corpus as well as having some more targeted text such as social media posts and news article text.

Data Set Cleaning

To get the maximum benefit from the above corpus we needed a method of processing and cleaning the datasets before they were used to train the classifier.

Three primary text processing libraries stood out as contenders for our project, they were ‘Spacy’ [13], ‘NLTK’ [14] and ‘Stanza’ [15]. When testing the libraries we wanted a balance of accuracy of functionality such as punctuation removal and stemming while maintaining performing. We timed the execution of the three libraries when stemming some text data.

The table below summarises our findings of stemming 500 sentences of average 20 words long using the different libraries:

Libraries	Spacy	NLTK	Stanza
Execution Time	1.36 Seconds	0.197 Seconds	34.33 seconds

Figure 1: Text Processing Libraries Performance Comparison

Spacy proved to be the most compelling solution for us. It had a smaller installation size compared to Stanza, and it was faster than Stanza. Comparing Spacy to NLTK, it was a little slower however its lemmatization functions were more accurate compared to NLTK that was very aggressive when it came to overstemming.

Classifier Training and Usage

Speaking to our project coordinator we realised we had three options to choose from when deciding how the classifier would be trained and then used. Each of the options below would train on the standard emotion classification datasets like the Unify dataset mentioned above.

- train the classifier on the cloud, and store the trained model on the client device. This model would then be applied to the user's text to classify the emotions in it. This allowed for the user's textual data to remain locally. However, the model would need to be lightweight enough so that it doesn't take up too much space on the users machine.

- train and run the classifier locally. For this to work, the training process shouldn't require too much memory so simple machine learning algorithms such as logistic regression or naive bayes could be used.

- train and run the classifier on the cloud. The major disadvantage with this approach is that the user's text would have to be stored in the cloud.

We consulted this research paper [16] and decided that we would have more flexibility training a classifier locally. Furthermore, the algorithms used by scikit learn to train a classifier would run efficiently on the hardware available to us, while not taking up too much storage space. By training locally we eliminated the need to manage cloud based services costs also.

Deducing The Emotions To Be Selected

We wanted our application to be able to classify at least five emotions. Researching psychological articles about core human emotions highlighted six basic emotions [17], with other emotions being subcategories of these. When selecting emotions it was important to match the emotions available in the datasets we had selected. We decided to select the emotions ‘anger’, ‘fear’, ‘joy’, ‘surprise’, happiness’ and ‘sadness’.

Program Operation

Due to the privacy concerns that some users may have regarding online browsing data we had to ensure that no browsing data would be transmitted externally from the users device. This meant that our application would be running locally on the users machine and that all classification would be run locally also. This influenced our decision in the section ‘Classifier Training and Usage’ above also.

Deciding on a GUI library

When deciding a GUI library there were many factors at play. We wanted a cross-platform library that would be aesthetically pleasing and not too difficult to learn. We had thought about using Javascript frameworks like ReactJS and VueJS for the frontend but decided against them as we wanted to focus solely on using Python. The idea of implementing the whole project in a Docker container seemed promising too.

However, we were adamant that a Python GUI would be practicable and we looked into various options like Tkinter [18], wxPython [19], and PyGUI [20]. While Tkinter is considered the de-facto Python GUI framework, it is very outdated and didn’t appeal to us. Furthermore we didn’t think it would have the flexibility to allow us to have the complex page structure that our application would need.

We ended up choosing PYQT5 [21] in the end as we had used applications built with the framework before like Calibre and Telegram. The C++ QT documentation was exhaustive and translated very well to PYQT. As well as this, creating UIs in PYQT5 was effortless and instant thanks to the QT Designer. The biggest downside when researching seemed to be the demanding learning curve.

Graphical Representation of Emotion Data

From our research, the Plotly library [22] seemed promising in being able to allow us to graphically display the emotion data. We wanted to use a selection of bar charts, pie charts, histograms and line plots to graphically represent browsing sentiment. We decided to keep plotly as our primary choice but remained open to other suitable libraries, such as the PyQt charts [23] library.

Final Research Decisions

When our research period was finished we had a clear understanding about the structure of our project.

Extracting History	URLs would be extracted from SQLite databases using SQL commands.
Text Scraping	Text scraping and rendering Javascript would be achieved with the scrapinghub/splash container in combination with BeautifulSoup to parse HTML documents and extract parts of the websites.
Text Processing	SpaCy would be used to parse text and lemmatize text. Some further removal of numbers, hashtags, and spelling errors would also be considered.
Emotion Classification	Emotion classification would be implemented using standard algorithms available in scikit-learn such as logistic regression and support vector machines. Time permitting a neural network approach may be considered. The classifier would be trained on data from the corpus discussed using the spacy library.
Frontend GUI & Visual Data	The application frontend would be implemented using PyQT5 and the QT Designer. The visualisation of data would be fulfilled by the most suitable plotting library.

System Design

Architecture Diagram

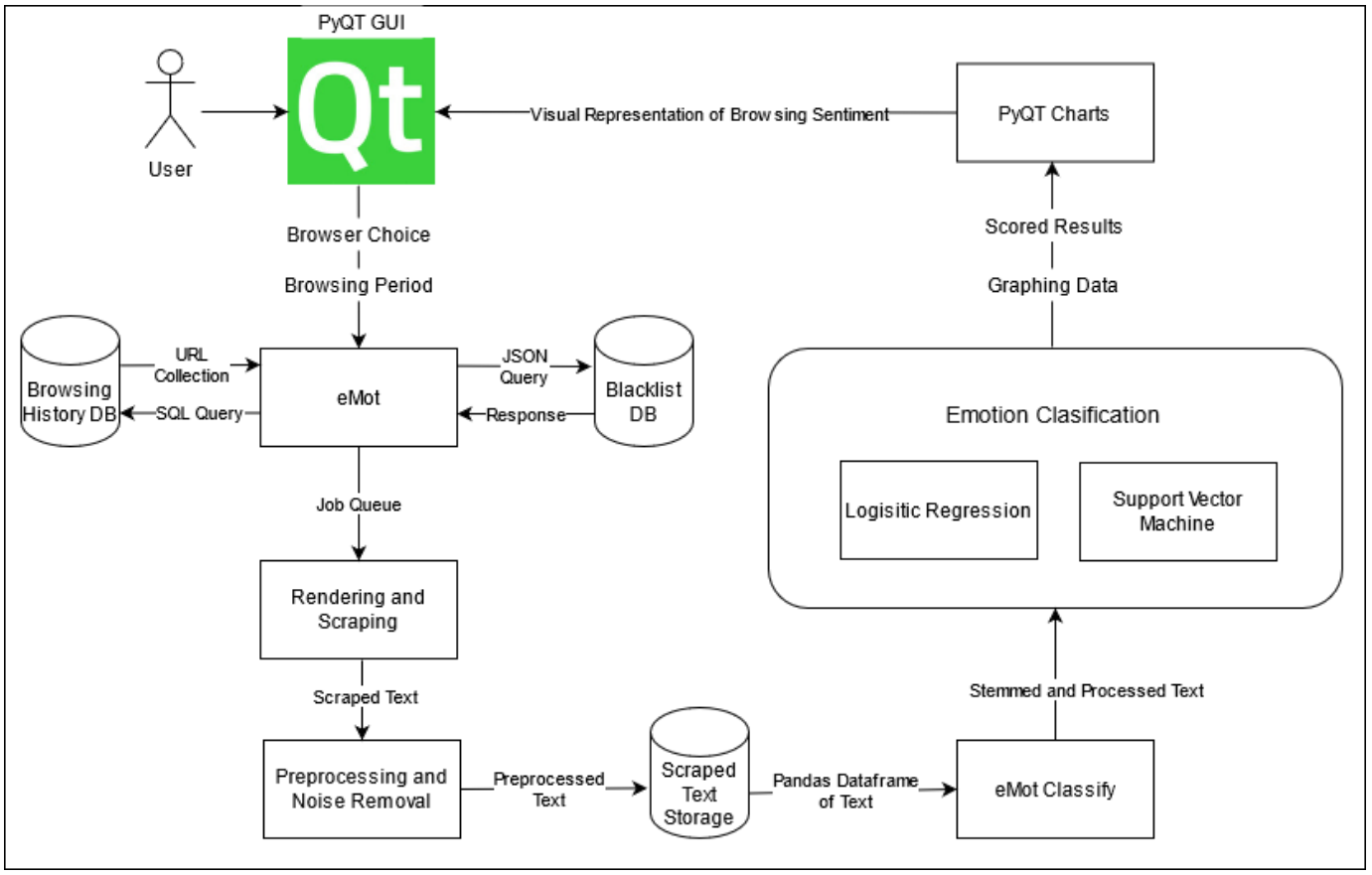

Figure 2: Architecture Diagram

This architecture diagram presents an abstraction of the software system. We can see the relationships, constraints, and boundaries between the components.

A notable comparison between this diagram and the architecture diagram in the functional specification is the ordering of the text scraping and text preprocessing. Originally, we had designed the diagram to preprocess parameters and then scrape text.

Sequence Diagram

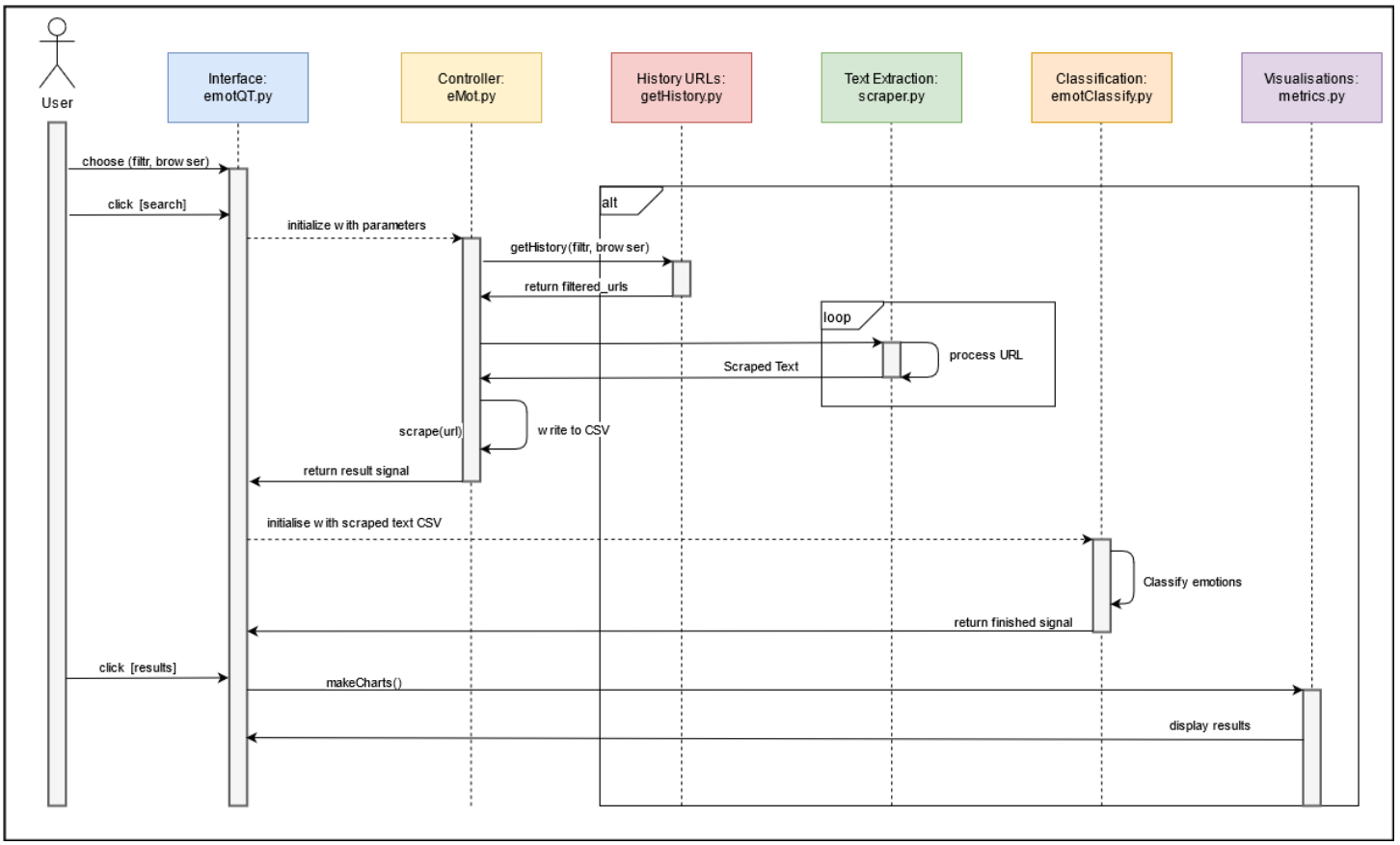

Figure 3: eMot Sequence Diagram

This sequence diagram shows the interactions of the main classes in the program between the interface, main controller class, history accessor, text extractor, classification class, and metrics visualisation class. The entry point for the user is the emotQT UI class.

The grey vertical bars show the period in which the class is in operation.The items inside the alt box represent a conditional execution. This represents the situation where a user does not have any urls returned from a search and the program finishes early. The loop box is executed multiple times until the queue full of URLs is empty.

Data Flow Diagram

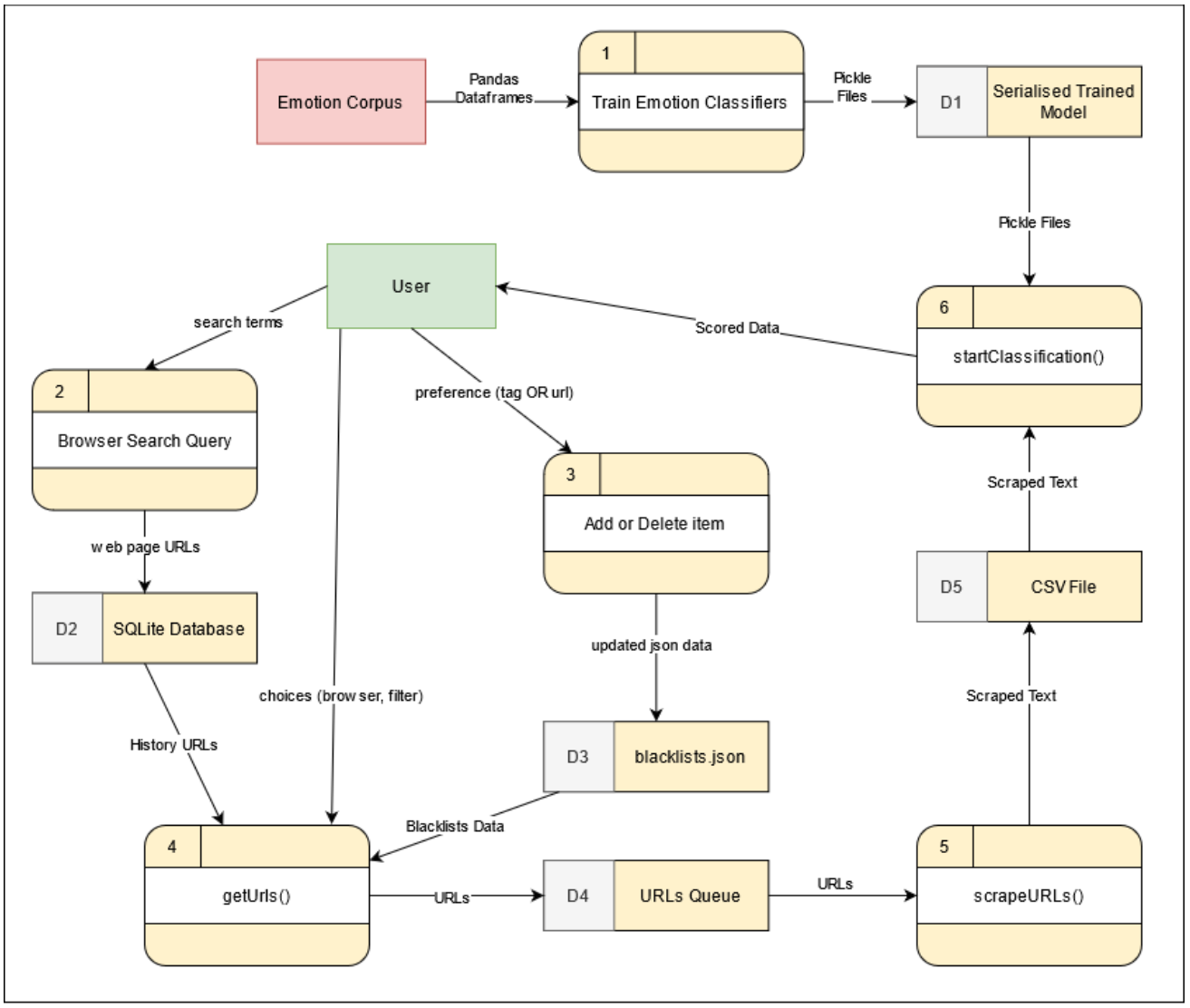

Figure 4: eMot Data Flow Diagram

The data flow diagram shows the manipulation and storing of data from external entities through the system. In this figure, we can see that the user adds web page URLs to the SQLite database whenever the user uses their browser to query search. This is done outside of the eMot application.

The user can add or delete tags or URLs to the backlists.json data storage through the preferences window of eMot. Whenever the user chooses their browser and filter choice in eMot, the data from the SQLite and Blacklist storage will be used to get URLs.

The Emotion Corpus entity sends datasets to be classified and trained. This data is stored in a serialized pickle file and waits as the data flows through the eMot application. When the scraped data is needed for emotion classification, the pickle files are loaded into the classifier and the final results data is presented to the user.

Implementation

URL Extraction

To access the browser history SQLite database we needed to execute SQL commands to select relevant articles from the SQL tables. We started by refreshing our SQL knowledge, starting with simple SQL commands and gradually building up more complex queries over time.

To allow our application to be cross platform we decided to adopt the 'browser-history' library [24], this provided the base code upon which we heavily expanded functionality. To allow the user to analyse specific browsing periods we needed to be able to compare dates from the history database. The method supplied by browser-history for working with dates was modified to use datetime objects using the datetime Python library.

Different browser options available for history extraction included Chrome, Firefox, Safari, Opera and some Chromium browsers such as Brave. Each browser required its own SQL command and predefined directory location where the SQLite database could be accessed by default. This behaviour is defined in the 'browserHistory' directory of our project.

```
windows_path = "AppData/Local/Google/Chrome/User Data"
```

Figure 5: Example Location Address

We also needed to consider that access to the SQL database would be locked when the internet browser would be in use. To overcome this we would make a temporary copy of the database to temp files and execute the SQL commands on the copy of the database. This way we would avoid blocking issues from file access issues with a small amount of overhead. This database copy would be stored in the operating systems temporary files registry.

Renderer and Text Extraction

In the proposal, we intended to use the Scrapy library as the system scraper. However, due to the complications previously expressed we eventually settled on Beautiful Soup for text extraction and Scrapinghub/Splash for rendering websites.

Scrapy was very precise with the types of HTML tags it would crawl the websites for and although this was a fast framework, we wanted to use a broad extract-all function.

Static websites can easily be scraped using any number of scraping libraries like Python requests, Beautiful Soup, Scrapy, Selenium, and LXML. Dynamic websites that use javascript rendering, on the other hand, need to be loaded in the DOM and then scraped. Many modern websites are using frameworks so we knew very early on that a static site scraper was out of the question [25].

We chose a combination of splash for rendering, Python requests for HTTP message passing, and Beautiful Soup for text extraction. As said previously, Splash is a headless browser that runs in a Docker container so we needed to use the request library to send the desired url to Splash using port 8050.

Datasets Amalgamation

Before we could use individual datasets from the corpus the files had to be pre-processed because some files consisted of different file structures and formatting. Some files were in text format where others were in CSV format. We decided to use CSV format because it allowed us to structure the data into rows and columns. When working with such large files we used the Pandas library to read in CSV files into dataframes which could be manipulated directly without having to read the CSV file to and from disk.

Some files consisted of columns such as author and date, these columns were discarded since they were of no use to us. We decided to keep a simple CSV file structure consisting of 2 columns ‘Emotion’ and ‘Text’ where emotion was one of the 6 chosen emotions and ‘Text’ consisted of textual examples portraying that emotion.

tweet_id	sentiment	author	content
1956967341	sadness	xoshayzers	@tiffanylue i know i was listenin to bad habit earlier and i started freakin at his part =[
1956977084	happiness	ktierson	mmm much better day... so far! it's still quite early. last day of #uds

Figure 6: Original CSV File Contents

Emotion	Text
sadness	@tiffanylue i know i was listenin to bad habit earlier and i started freakin at his part =[
happiness	mmm much better day... so far! it's still quite early. last day of #uds

Figure 7: Altered File Example

When we had selected the datasets from the corpus we were going to use, we started to extract the 6 emotions we had selected from these datasets. All occurrences of each emotion were placed in their own CSV file along with all other examples of the same emotion. At the end of this process we had 6 CSV files consisting of labelled examples of the emotions from different datasets in one file with a common file structure.

Dataset Cleaning

The files created in the ‘Datasets Amalgamation’ stage then had to be altered before they could be used in training a classifier. Different preprocessing tasks were carried out on the chosen datasets, including changing all text to lowercase, spell checking, URL removal, hashtag and symbol removal. Additionally tokenization, punctuation removal, stop word removal and spell checking were carried out.

Text Modification

For tasks such as URL, hashtag, symbol and digit removal the Python regex library was used. Regex patterns were defined to match patterns of charters in text and remove them.

Tokenization
For tokenization of text we used the spacy library. Spacy's tokenizer broke the sentence structures into individual tokens and tagged the words in the process using the words semantics.

Stop Word Removal
Spacy was also used for stop word removal. Using the tags on the words during the tokenisation stage allow us to check if a word was a stop word. Since these common terms would not help in classifying emotions and didn't carry much meaning in the sentence they were removed.

Lemmatisation
Spacys lemmatizer was used to stem words to their base form. We chose lemmatization over stemming because it uses the word in context based on part of speech tagging [26]. While this process did perform a little slower than stemming, we felt that the decreased accuracy of a standard stemming approach would be detrimental to the emotion classification accuracy of the system overall.

Emotion	Text
sadness	i know i was listening to bad habit earlier and i started freakin at his part
happiness	much better day so far it is still quite early last day of

Figure 8: Example of cleaned sentences from figure 7

Emotion Classifier

For this project we wanted to include a mix of different classifiers to get different 'views' of the text data. We planned on using diverse classifiers and seeing if they agreed on the classification. This would allow us to compare outputs from both classifiers before accepting a result.

Primary research led us to logistic regression classifiers. We started with a binomial classifier that scored text as either negative or positive. We then expanded this into a multinomial classifier by adding a third 'neutral' class. For the remainder of our project this would act as our 'baseline' which we could compare progress to. After further investigation we expanded the Logistic regression (LR) classifier to use stochastic gradient descent fitting.

When analysing our options for the second classifier we decided to incorporate a support vector machine (SVM), specifically a linear SVM (LSVM) [27], This is widely regarded as one of the best text classification algorithms. Both LSVM and SGD classifiers were hyperparameter tuned using the grid search options available in scikit [28].

Document term matrix usage/Feature Extraction
The cleaned and tokenized words were put in a document term matrix, the two options used by use were (i) count vectoriser creation and (ii) tf x idf calculation

Count Vectoriser: The count vector or bag of words encoding scheme was used in conjunction with the Logistic regression model to constitute word vectors for a given document.

Term frequency Inverse Document Frequency (TFIDF): TF-IDF was used for the SVM classifier. TF shows the frequency of a term in a given document and IDF is used to calculate the frequency of the word in the total number of documents. This allows for the weighting of specific words to be altered given their frequency.

Trained models were saved using the pickle library. This allowed us to save the trained classifier and use the classifier to score text at a later date.

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 [29]. To check the balance of emotion scoring accuracy across the six emotions a confusion matrix was printed after each train [30]. This allowed us to view if some emotions were being confused for other emotions.

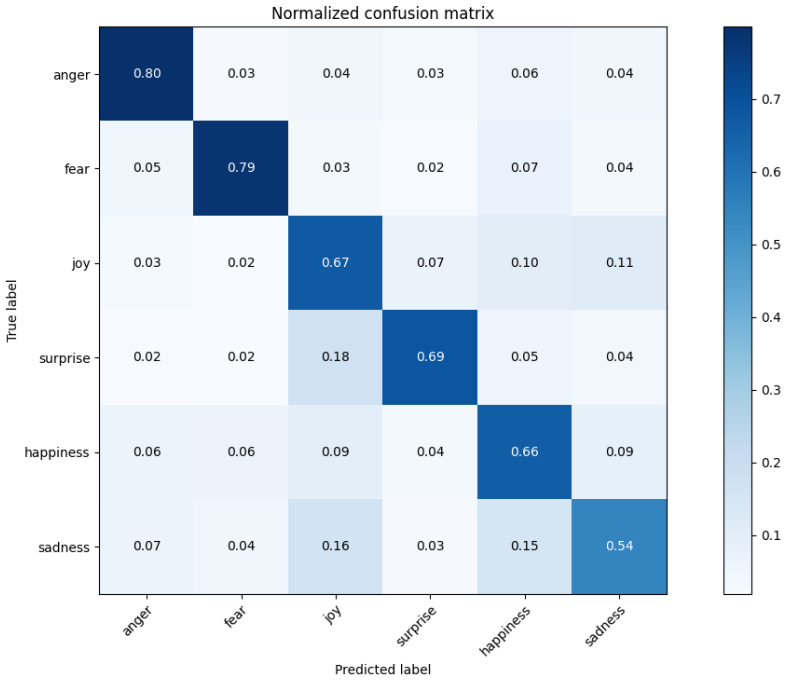

Figure 9: An example Confusion Matrix

Web Scraper

Our early implementations of a web scraper involved the requests library and Beautiful Soup serially traversing through a list of urls. As you might expect this was exceptionally slow and progressed through the list in a synchronous manner, executing a new scrape only when the previous one had finished. We knew that threading would be useful for speeding this process up. For our initial implementation of a multithreaded version of the scraper, we manually created a thread for each URL extracted from the browser history database. We soon learned that this was an expensive approach to threading due to the overhead associated with creation and deletion of individual threads.

We therefore decided to turn our focus to implementing some concurrency design patterns we had been learning about in the CA4006 module. Having analysed the different design patterns available we decided that the use of a Master-Worker pattern with a thread pool would be the most suitable solution. Using the futures library we were able to implement a fixed size thread pool [31].

Our choice was influenced by the following:

- It was impractical to assume 1 thread per task.
- Each time the scraper is run there is a dynamic number of sites to scrape.
- Scraping and rendering tasks are embarrassingly parallel.
- Thread pools are good for load balancing.
- The thread pool allows us to treat tasks as equal even if different sites take different times to process, we can allocate them as if they are all the same.
- A thread pool allows us to limit the number of threads, to avoid bottlenecking the users system.
- Urls are queued and while there are still items on the queue threads are continually fed work to do.
- Scraping relies on I/O and this can be very slow.

When a thread finishes its work it returns to the pool and may get reallocated work if URLs are still present on the scraping queue. Fortunately to support multiple threads the 'splash' renderer is stateless and can scale to multiple instances so we encountered no problems here.

```
72          with futures.ThreadPoolExecutor(max_workers=MAX_WORKERS) as executor:
73              threads = []
74              i = 1
75              while not queue.empty():
76                  url = queue.get()
77                  threads.append(executor.submit(Scraper().scrape,url=url,task=i))
78                  i += 1
79
80              # Deal with the threads as they complete individually
81              for future in futures.as_completed(threads):
82                  # tuple
83                  scrapedText = future.result()
84
85                  if scrapedText is not None:
86                      url = scrapedText[0]
87                      text = scrapedText[1]
88                      # store scraped scrapedText
89                      self.writeToCSV(url, text)
```

Figure 10: snippet of async execution of tasks

This approach to parallelism allowed us to execute asynchronous tasks and as jobs completed the result of the scrape was obtained from the futures object and then stored in a CSV file.

Classification of Text

When deciding if we would use a document or Sentence level classification to score textual data we had to consider the length of text documents that would be classified. We decided to select sentence level classification since document level would be more suitable to shorter documents, and text sites that would be scrapped would likely be too large to get an accurate result using it.

Graphical Representation of Sentiment Data

The GUI stage of eMot required a lot of rework as we did not decide early which of the three main plotting libraries would be suitable for the project. They all had equal strengths and weaknesses. These libraries were the native PyQtChart bindings, PyQtGraph, and Plotly.

While we had initially thought that Plotly would be suitable for this project, we soon discovered however that integrating Plotly into our PyQt GUI could prove difficult since we would need to handle an internal web browser to showcase the graphs.

PyQtGraph widgets were not difficult to integrate but the library was more suited for scientific and mathematical applications. The graphs were not completely appropriate for the eMot data so they were eventually discarded in favour of PyQtCharts.

PyQtChart did not have as much thorough functionality as the other libraries but the customizability and animations were great for eMot. One beneficial aspect of this native library was the fact that components in the QT Designer could be promoted to PyQtChart components before appending any data. This allowed us to have full control over the layout of these charts by scaling and rearranging them comfortably.

Sample Code

Redirecting output

```
72      def setupPrintPage(self):
73          self.stackedWidget.setCurrentWidget(self.printPage)
74
75          # Stream directs output to the textEdit box.
76          sys.stdout = Stream(newText=self.redirectText)
```

Figure 11: setupPrintPage function snippet in emotQT.py

In this function, Sys.stdout is set to an instance of the Stream class whenever the go button is pressed.

```
175     class Stream(QtCore.QObject):
176         """Redirects console output to text widget."""
177
178         newText = QtCore.pyqtSignal(str)
179
180         def write(self, text):
181             self.newText.emit(str(text))
```

Figure 12: Stream class in emotQT.py

The Stream class emits a string of the passed text.

```
85      def redirectText(self, text):
86          # Write console output to textEdit widget.
87          cursor = self.textEdit.textCursor()
88          cursor.movePosition(QtGui.QTextCursor.End)
89          cursor.insertText(text)
90          self.textEdit.setTextCursor(cursor)
91          self.textEdit.ensureCursorVisible()
```

Figure 13: redirectText function in emotQT.py

This function shows the stream where to emit new strings. The cursor is always set to the end of the textEdit, which will be the output in the terminal.

PYQT Worker class

```python
    def __init__(self, fn, *args, **kwargs):
        super().__init__()
        # Store constructor arguments (re-used for processing)
        self.fn = fn
        self.args = args
        self.kwargs = kwargs
        self.signals = WorkerSignals()

    @pyqtSlot()
    def run(self):
        # Initialise the runner function with passed args, kwargs.
        # Retrieve args/kwargs here; and fire processing using them
        try:
            result = self.fn(*self.args, **self.kwargs)
        except:  # noqa
            # traceback.print_exc()
            exctype, value = sys.exc_info()[:2]
            self.signals.error.emit((exctype, value, traceback.format_exc()))
        else:
            # Return the result of the processing
            self.signals.result.emit(result)
        finally:
            self.signals.finished.emit()  # Done
```

Figure 14: Worker class in qtworker.py

The Worker class processes function, args, and kwargs. It then emits a pyqtSignal() when it is finished or returns a result. An instance of this class was passed to the threadpool where it would run without freezing the program.

QT Designer

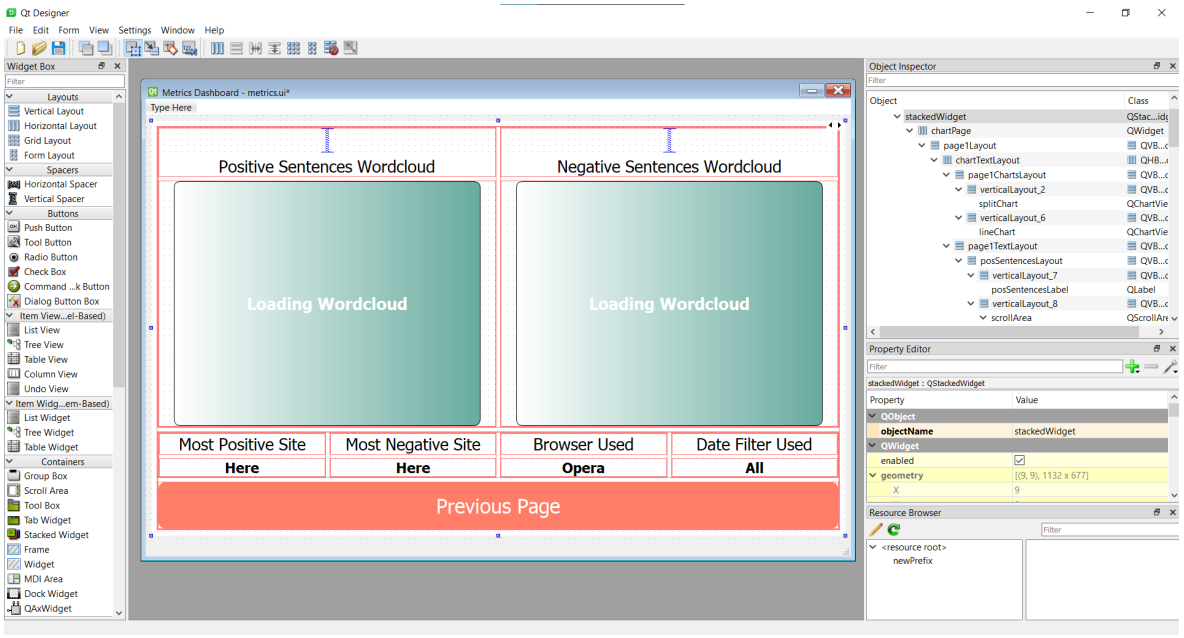

Figure 15: The wordcloud metric page UI created in QT Designer

The QT Designer is a drag and drop interface for creating quick user interfaces. Every object in the PYQT system is inserted into a layout to maintain a scalable window. In this screenshot, there are 24 nested layout widgets. The Designer offers no functionality but instead can be inherited by a super class and given functionality.

Problems Solved

Tailoring the Scraper to the System

Because the application will have different resources available to it depending on the system on which it runs we needed to tailor the scraper to match these resources. Using the multiprocessing Python library we were able to query how many cores were present on the PC. An upper bound of threads allocated to the scraper would be double the core count of the system. So for example a system with 6 cores the scraper would be allocated a maximum of 12 threads, whereas on a dual core machine a maximum of 4 threads would be created. While this may slow down the scraping process it would prevent the overall application lagging from the scraper bottlenecking resources.

```
cpu_cores = multiprocessing.cpu_count()
MAX_WORKERS = cpu_cores * 2
```

Figure 16: allocation of max_workers based on cpu count

The parameter 'max_workers' as seen in this image is calculated according to the number of cores available in the system the application is running on.

```
with futures.ThreadPoolExecutor(max_workers=MAX_WORKERS) as executor:
```

Figure 17: threadpool executor using max_workers

Balancing Datasets

To prevent over or underfitting certain emotions when training the sentiment classifier we had to ensure that the dataset we used was balanced with the emotion counts. We used the Pandas library to analyse the corpus to check for value counts. Many of the original datasets from the corpus were unbalanced with some emotions having nearly double the count of other emotions.

This required us to amalgamate different files together to increase the count for some emotions as described in the section 'Datasets Amalgamation' above. From the available datasets from the corpus we were able to build 6 datasets for the 6 emotions we had selected. Each dataset had on average 5000 labelled sentences.

These datasets were then stored on Google Drive to prevent the Gitlab Repo from growing unnecessarily large. The customised datasets can be viewed at this link. The files chosen from the corpus [32] are described in the following table:

Dataset	**Contents**
dailydialog	Manually labelled human conversations dataset.
isear	Reported situations in which emotions were experienced.
emoint	Annotated datasets of social media posts.
crowdflower_data	Annotated dataset of tweets via crowdsourcing.

Figure 18: Dataset and contents description

Slow Automated Tests

Since integrating Gitlab CI into our project we saw how long it took to run automated tests on the Gitlab Runner server. To overcome this we decided to create a Docker image to package application dependencies inside a Python image. This image would then be used for testing, instead of installing the project dependencies each time tests were run we just used the environment created in the docker image.

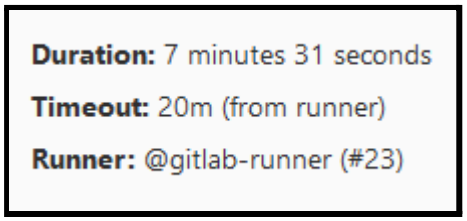

Figure 19: Testing without Docker Image

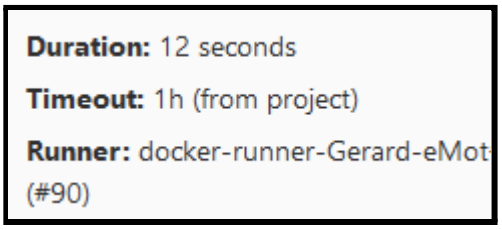

Figure 20: Testing with Docker Image

This significantly decreased the testing times as the image was cached for future tests to use. However, near the end of the project we had to roll back to the old method of testing when the gitlab runner machine began to run out of storage space, causing some of our automated tests to fail.

Redirecting stdout to PYQT

Before applying the program to a GUI, eMot was run in the terminal with results being printed to the terminal only. These print statements were very useful so we wanted to replicate a terminal output on eMot. Initial redirection worked but all the text would print in one block text after completion as it was taking up resources and freezing. The solution was to add a PYQT QThreadpool that could handle any standard out text in real-time. We used a worker class that inherited QRunnable and could be passed the function name with additional args and kwargs. The worker was added to a QThreadPool and could handle any function callback.

Successfully adding the QThreadpool was a major achievement as it allowed us to then improve the performance in other areas like creating the statistics and wordclouds without the main application freezing.

Reading the Scraped Text CSV Once

When we started to classify scraped text from the CSV file we realised we were loading the file twice into two separate pandas dataframes. To overcome this we extracted the csv read logic into its own function. Any alteration of the dataframe is carried out in this function.

Some examples of altering the data frame are creating a column for the ‘base’ url. The base url consists of the domain name and suffix. The scraped text is stemmed and cleaned further in this dataframe for classification of emotions.

Having a copy of the original text and stemmed text in the dataframe allowed us to classify the stemmed text but give sentence examples in the PyQT GUI of the original unaltered text.

```
84      def readScrapedFile(self):
85          scraped_df = pd.read_csv(
86              self.scrapedFile).astype("U")
87
88          # create a new column with the base baseUrl as its value
89          scraped_df["base"] = scraped_df["url"].apply(base)
90          # create a list of unique base sites
91          sitesList = scraped_df["base"].unique().tolist()
92
93          # stem the data for classification
94          scraped_df["stemmedText"] = scraped_df["text"].apply(clean)
95
96          # create a nested dictionary for each site
97          for site in sitesList:
98              # dictionary key maps to a nested dictionary
99              self.emotionsPerSite[site] = copy.deepcopy(self.emotionsDict)
```

Figure 21: altering Panda dataframe columns

Testing

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.

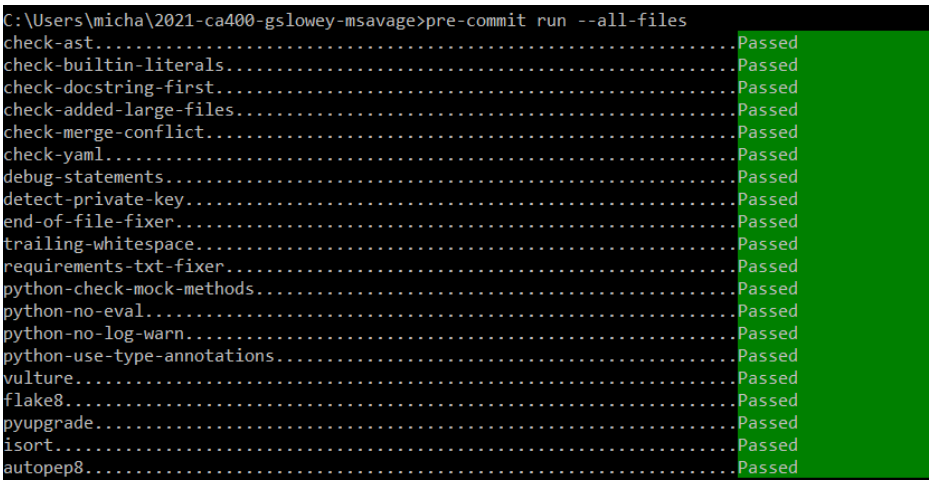

Figure 22: Manual run of pre-commit showing tests passing

Unit Testing

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.

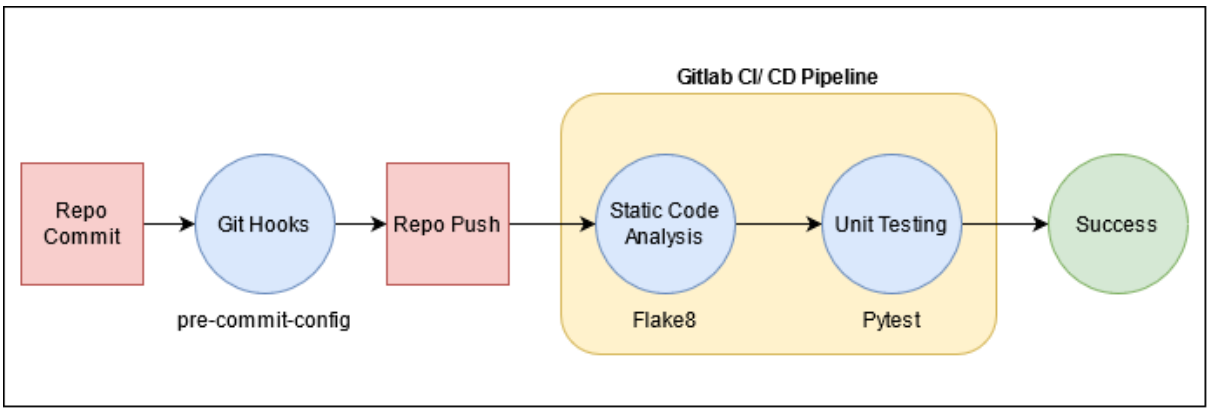

Figure 23: Unit Testing Flow

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 results of the user testing feedback are available here.

Learning Experiences

Self Reflection - Michael Savage

- I believe I became really proficient at using Git through trial and error. I have learned some valuable skills for maintaining a shared repository and think these skills will stick with me in industry.

- I developed a clear understanding of PYQT and PYQT signals after a demanding learning curve. This had a knock-on positive effect on my understanding of Javascript frameworks for another module this year. I generally would have shied away from frontend development but I feel very confident in my abilities now.

- Going into the project I thought I had a good grasp of Python. I was quite wrong on this assessment and had to really take time to understand some data structures. I feel like I have become quite adept at using Python and taking advantage of it. Two interesting concepts I enjoyed using were f strings and lambda functions

Self Reflection - Gerard Slowey

This project in my opinion involved a high level of technical difficulty and complexity. At times the project required some rework due to reduction in performance from poor implementation. However we constantly had an eye for software quality and constant refactoring prevented the system from growing in complexity towards an unmaintainable nightmare.

I believe I strengthened my Python skills considerably during the course of the project and used some libraries that previously I had never used before the most important ones being Pandas and Scikit Learn.

- Pandas proved to be invaluable in data manipulation using dataframes. Dataframes were used while training the classifiers, manipulating the datasets and reading the scraped data. The overall library usability and performance were vital in the deadline with large CSV files.

- Scikit Learn was quite a steep learning curve but allowed me to introduce a machine learning aspect to the project with ease. Again the usability of the library supported by the extensive documentation were very beneficial for expanding my understanding while implementing new functionality.

- Like Michael I too refined my use of Git and Gitlab, I learned how to properly fix merge conflicts and how Gitlab is so much more than a source control system with tools such as CI runners. By incorporating tools and automation into the project we reduced the need to carry out repetitive and mundane tasks.

Future work

Improving Classifier Accuracy

Towards the end, when the project was near completion we investigated the usage of a neural network for emotion classification. We looked at Tensorflow and PyTorch.

For future work we would plan to research into LSTM and Bert. From reading we saw the advantages of using pre-trained word vectors to be able to better classify unseen text and how the use word semantics can be used to calculate relations and similarities between words.

Throughout this project we were very much aware that emotion classification is still a developing sector with much ongoing research. Emotion detection from online content is relatively a new and challenging area in computational intelligence attracting attention of researchers in the recent past.

Improvement of Graphing Strategy

Our application's classifying functionality deals with data science patterns. This is a completely different branch of software engineering that we did not have much experience with. The system could therefore be enhanced by researching data science techniques and learning how to apply data analysis and complex graphs.

Packaging the System

Finally, one major aspect of the program is the installability of it. It was unfortunate that we could not package the program into a single executable file. Going forward, this would be a key area of interest because the program currently needs Python and Docker Desktop to be installed. This is not an accessible solution for a general audience.

References

[1] J. Wagner *et al.*, “DCU: Aspect-based Polarity Classification for SemEval Task 4,” in *Proceedings of the 8th International Workshop on Semantic Evaluation (SemEval 2014)*, Dublin, Ireland, 2014, pp. 223–229, doi: 10.3115/v1/S14-2036.

[2] M. Z. Asghar *et al.*, “Performance Evaluation of Supervised Machine Learning Techniques for Efficient Detection of Emotions from Online Content,” p. 30.

[3] “How to Create a Chrome Extension in 10 Minutes Flat - SitePoint.” https://www.sitepoint.com/create-chrome-extension-10-minutes-flat/ (accessed May 02, 2021).

[4] “DB Browser for SQLite.” https://sqlitebrowser.org/ (accessed May 02, 2021).

[5] R. Taracha, “Introduction to Web Scraping using Selenium,” *Medium*, Apr. 27, 2018. https://medium.com/the-andela-way/introduction-to-web-scraping-using-selenium-7ec377a8cf72 (accessed May 02, 2021).

[6] R. Python, “Beautiful Soup: Build a Web Scraper With Python – Real Python.” https://realpython.com/beautiful-soup-web-scraper-python/ (accessed May 02, 2021).

[7] “Scrapy | A Fast and Powerful Scraping and Web Crawling Framework.” https://scrapy.org/ (accessed May 02, 2021).

[8] “Splash - A javascript rendering service — Splash 3.5 documentation.” https://splash.readthedocs.io/en/stable/ (accessed May 02, 2021).

[9] “configparser — Configuration file parser — Python 3.9.4 documentation.” https://docs.python.org/3/library/configparser.html (accessed May 02, 2021).

[10] G. Qiu, B. Liu, J. Bu, and C. Chen, “Opinion Word Expansion and Target Extraction through Double Propagation,” *Computational Linguistics*, vol. 37, no. 1, pp. 9–27, Mar. 2011, doi: 10.1162/coli_a_00034.

[11] “A Survey and Experiments on Annotated Corpora for Emotion Classification in Text | Institut für Maschinelle Sprachverarbeitung | Universität Stuttgart.” https://www.ims.uni-stuttgart.de/forschung/ressourcen/korpora/unifyemotion/ (accessed May 02, 2021).

[12] L. A. M. Bostan and R. Klinger, “An Analysis of Annotated Corpora for Emotion Classification in Text,” p. 16.

[13] “spaCy · Industrial-strength Natural Language Processing in Python.” https://spacy.io/ (accessed May 02, 2021).

[14] “Natural Language Toolkit — NLTK 3.6.2 documentation.” https://www.nltk.org/ (accessed May 02, 2021).

[15] “Overview,” *Stanza*. https://stanfordnlp.github.io/stanza/ (accessed May 02, 2021).

[16] S. Dhar, J. Guo, J. Liu, S. Tripathi, U. Kurup, and M. Shah, “On-Device Machine Learning: An Algorithms and Learning Theory Perspective,” *arXiv:1911.00623 [cs, stat]*, Jul. 2020, Accessed: May 02, 2021. [Online]. Available: http://arxiv.org/abs/1911.00623.

[17] “Our Basic Emotions Infographic | List of Human Emotions,” *UWA Online*. https://online.uwa.edu/infographics/basic-emotions/ (accessed May 02, 2021).

[18] “tkinter — Python interface to Tcl/Tk — Python 3.9.4 documentation.” https://docs.python.org/3/library/tkinter.html (accessed May 02, 2021).

[19] T. wxPython Team, “Welcome to wxPython!,” *wxPython*, Dec. 24, 2019. https://wxpython.org/index.html (accessed May 02, 2021).

[20] J. H. and P. Cothren, *dearpygui: DearPyGui: A simple Python GUI Toolkit*. .

[21] R. C. Limited, *PyQt5: Python bindings for the Qt cross platform application toolkit*. .

[22] “Plotly: The front end for ML and data science models.” / (accessed May 02, 2021).

[23] R. C. Limited, *PyQtChart: Python bindings for the Qt Charts library*. .

[24] S. Sarnayak, *browser-history: A python module to extract browser history*. .

[25] A. Jones, “How to Scrape Dynamic Web pages with Selenium and Beautiful Soup,” *Medium*, Apr. 22, 2021. https://towardsdatascience.com/how-to-scrape-dynamic-web-pages-with-selenium-and-beautiful-soup-fa593235981 (accessed May 02, 2021).

[26] “Stemming and lemmatization.”

https://nlp.stanford.edu/IR-book/html/htmledition/stemming-and-lemmatization-1.html (accessed May 02, 2021).
[27] “1.4. Support Vector Machines — scikit-learn 0.24.2 documentation.” https://scikit-learn.org/stable/modules/svm.html#svm (accessed May 02, 2021).
[28] “3.2. Tuning the hyper-parameters of an estimator — scikit-learn 0.24.2 documentation.” https://scikit-learn.org/stable/modules/grid_search.html (accessed May 02, 2021).
[29] “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).
[30] “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).
[31] “concurrent.futures — Launching parallel tasks — Python 3.9.4 documentation.” https://docs.python.org/3/library/concurrent.futures.html (accessed May 02, 2021).
[32] “sarnthil/unify-emotion-datasets,” *GitHub*. https://github.com/sarnthil/unify-emotion-datasets (accessed May 02, 2021).