H446 Component 3 Programming project

Contents

Analysis

Problem Identification

Twitter is a platform that facilitates the expression of the opinions of millions with hundreds of millions of tweets being made every day. With this large volume of data, it can become problematic for companies to try and navigate through this data to find information relevant to them. This may include twitter users tweeting directly about the company or other potential opportunities and business insight Twitter can provide. Sentiment analysis offers a solution to this problem, providing automated and valuable insights into public opinions on various topics.

Sentiment analysis is a branch of Natural Language Processing, allowing a computer to competently identify human emotions expressed in text. In this instance, the sentiment analysis system would specialise in the analysis of Twitter tweets, allowing the user to gain an understanding of the consensus of their company, including their strong attributes and areas for improvement that have been voiced outside of formal review. It also allows companies look at the opinions on their competitors as well as enabling them to look at online trends that they could capitalise on. Without the use of sentiment analysis, companies looking to find such data would need to expend resources

searching for and cataloguing these opinions, which makes the use of sentiment analysis an appropriate approach.

Why it is suited to a computational solution

There is an obvious need for a computational approach as the system is accessing twitter data which is available online and so a computational approach is certainly a necessary feature of the solution.

This problem is amenable to computational approach through its need for automation which would not be possible without a computer. The ability of a computer to automate the process saves the company time by not requiring them to search through numerous twitter feeds and collate relevant information. A computational approach also allows for accurately kept up-to-date analyses. This can be done through using databases as a persistent store of analysis results, eliminating the need for users to create physical records of analysis data which would also waste company resources.

A major benefit of using a computational approach to the solution is the use of machine learning and the technology's scalability. Without this technology, it would become too difficult for the company to track opinions expressed on Twitter manually due to the volume of tweets being produced and the need to physically record the content of these tweets. However, through using machine learning, an increase in tweet volume will only slow down performance of the system by seconds as oppose to the hours it would take to handle this data manually.

Computational methods the solution accommodates:

Abstraction

Sentiment Analysis uses abstraction by nature due to the polarity of the output provided by the system. Typically, a system using sentiment analysis will only describe text as being positive, negative, or neutral (which is why it is described as polar). This is a clear abstraction of a tweet's intention and purpose as text is not normally wholly positive or negative or neutral. For example, according to the software, two different tweets about a product one saying it is "excellent" and the other saying it is "quite good" will both be placed in the "positive" category despite the two phrases conveying different amounts of positivity. This reduction in complexity makes large quantities of data more digestible for a human user and more manageable for a computer system.

Decomposition

There are two main features of the proposed system, the machine learning aspect and the core system which utilises the machine learning model to provide information for the user. The solution can be broken down into two primary elements, the machine learning model, and the programs that makes up the system. These primary elements of the system can be further broken-down using decomposition as follows:

The core ability of the machine learning model

1. Receive text as input.
2. Read the text and make a prediction on the sentiment of the text.
3. Return the prediction of the sentiment of the text.

The core components of the working system

1. Scrape tweets about the company and any topics that may be relevant to the company
2. Perform sentiment analysis on these tweets using the machine learning model
3. Provide a graphical user interface to display results
4. Make use of a database to store analyses results

Visualisation

Visualisation will be used for the user's benefit with graphs and charts being created based on the results of sentiment analysis. This visual aid will help the user understand the large quantities of data that the system will be handling.

I will also be using visualisation for my own benefit during development as using graphs allows me to compare the performance of different models so I can find the model that performs the best on the test data.

Performance Modelling

I will need to perform performance modelling on the machine learning model I need to create for the system so that it is as accurate as possible. By carrying out performance modelling on test data, I can ensure that the system is providing accurate results on what people are tweeting about otherwise the software becomes redundant.

Data Mining

Data mining plays a major role within any kind of sentiment analysis. Data mining in this instance involves the component of the system that scrapes tweets from Twitter. Sentiment analysis is itself a form of data mining, also being referred to as "opinion mining".

Clients and Potential Stakeholders

The primary stakeholders for this software would be businesses looking for an insight into public opinion on their company which may help them with their business strategies. My specific client for this software will be Andrew Johnson, who works in marketing at BlueWater Games, a videogames publisher. Despite BlueWater's success, they are struggling to handle the volume of consumer feedback. I will conduct an interview with my client to discuss potential features of the final solution as well as confer with him about the designs of the solution.

This kind of software is suited to large businesses like BlueWater where many people are tweeting about them and offering their opinions. As soon as a major company like BlueWater releases a new game, many Twitter users are quick to express their excitements and criticisms like graphics or cost. Normally, it would take a large team to handle this large number of tweets, but this proposed system can take this role instead. By understanding consumer's thoughts on their company and their new games, the company can then work to improve this new release and/or adjust their strategy for future releases.

Large and small businesses alike could also use the software as means to carry out research on what consumers are saying about competitors and more successful businesses as well as be able to identify trends in the market that they could capitalise on. In the case of my client, BlueWater may look at how consumers view other games publishers and deduce whose products consumers prefer and why. In terms of market trends, my client could use the software to identify the videogame that is currently most popular. BlueWater could then use this data to make informed decisions when developing their next releases.

Problem Research

Interview with client

Above, I have described what the fundamental purpose of the proposed system should be. To explore the more specific functionality of the system, I will carry out product research involving conducting an interview with my client and looking at existing solutions.

Questions Overview

To find out how to meet the specific needs of the client. I have created multiple interview questions that will give me an insight into the best approach for creating the solution. The questions I will ask my client are as follows.

1. **What is currently your primary source of feedback for your releases?**

This is so I can grasp to what extent they are considering the feedback of the consumers themselves.

2. **What is currently your primary method of receiving feedback for your releases?**

This question is necessary to assess what their preferred approach is to receiving feedback and look at whether this process is automated.

3. **Are you currently using Twitter in any capacity as means to receive feedback from customers?**

This question is to assess the extent to which the company is already using Twitter insights.

4. **What kind of specific insights would you like be provided by the software?**

This question is specifically, asking the client about how he would like the data to be presented and specifically what data to display for the user. This is important so I know specifically what kind of data I should display for the user.

5. **Do you have any further requirements for the software?**

This is asked so I can address any other requirements not addressed by the questions.

Interview

1. **What is currently your primary source of feedback for your releases?**

"Currently, we are focussing on looking at formally written reviews of our games found on our review sites. This is because generally these reviewers know what they are talking about and tend to be more familiar with what modern games fundamentally require. We only have a small team taking in feedback for our releases and so we are limited to just looking at these reviews."

2. **What is currently your primary method of receiving and working upon feedback for your releases?**

"Normally, for each release we just go through a list of reputable review sites we have received feedback from over the years and the team makes note of any significant critiques in these reviews. Often, action is only taken to amend these issues if they are really significant which is something I'd like to change"

3. **Are you currently using Twitter in any capacity as means to receive feedback from customers?**

"Because of the small size of the team, we can't really efficiently make use of Twitter at the moment, but I recognise it is definitely a platform I am looking forward to be able to use in future due to the popularity of the site. The issue with Twitter for us is the time it takes to look through the site and find tweets that actually provide meaningful feedback for us."

4. **What kind of specific insights would you like be provided by the software?**

"It would be useful to see changes in the opinions of the company over time. This would help us grasp how well our customers have responded rather than just purely looking at product sales. It would be helpful to also be able to just have a page where we can read some of the tweets themselves and it would be even better if there was a way the software could provide some of the most common things brought up in tweets about the company and our games."

5. Do you have any further requirements for the software?

"*It would also be good if there was some kind of way to save and view the results externally, so the data isn't limited to the software to make it a bit easier to work with the results."*

Takeaways from client interview

Considering the interview with my client, I need to make sure the use of twitter data is useful for them given that they are already handling feedback from other sources. This can be done by having data that is specific to Twitter such as displaying which hashtags were used and what some of the most popular tweets about the company were saying.

My client also said that currently, they are unable to use Twitter for feedback due to the small size of the team. I inferred from this that the team could benefit from an emphasis on automation within the software to maximise their efficiency.

Looking at his requirements for the insights, they would benefit from the software having a feature where it logs data over time. This would be straight forward to implement by using a database. As for displaying the tweets, this should be an easy feature to add as I will have to retrieve the tweets for analysis anyway so I should be able to display them and can then display them in relation to their sentiment predicted by the model.

My client also suggested the ability to save and view the data outside of the software which I suppose would help in sharing the results with others. As I will use a database to store the results of any analyses, this should also be straight forward to implement.

Existing Solutions

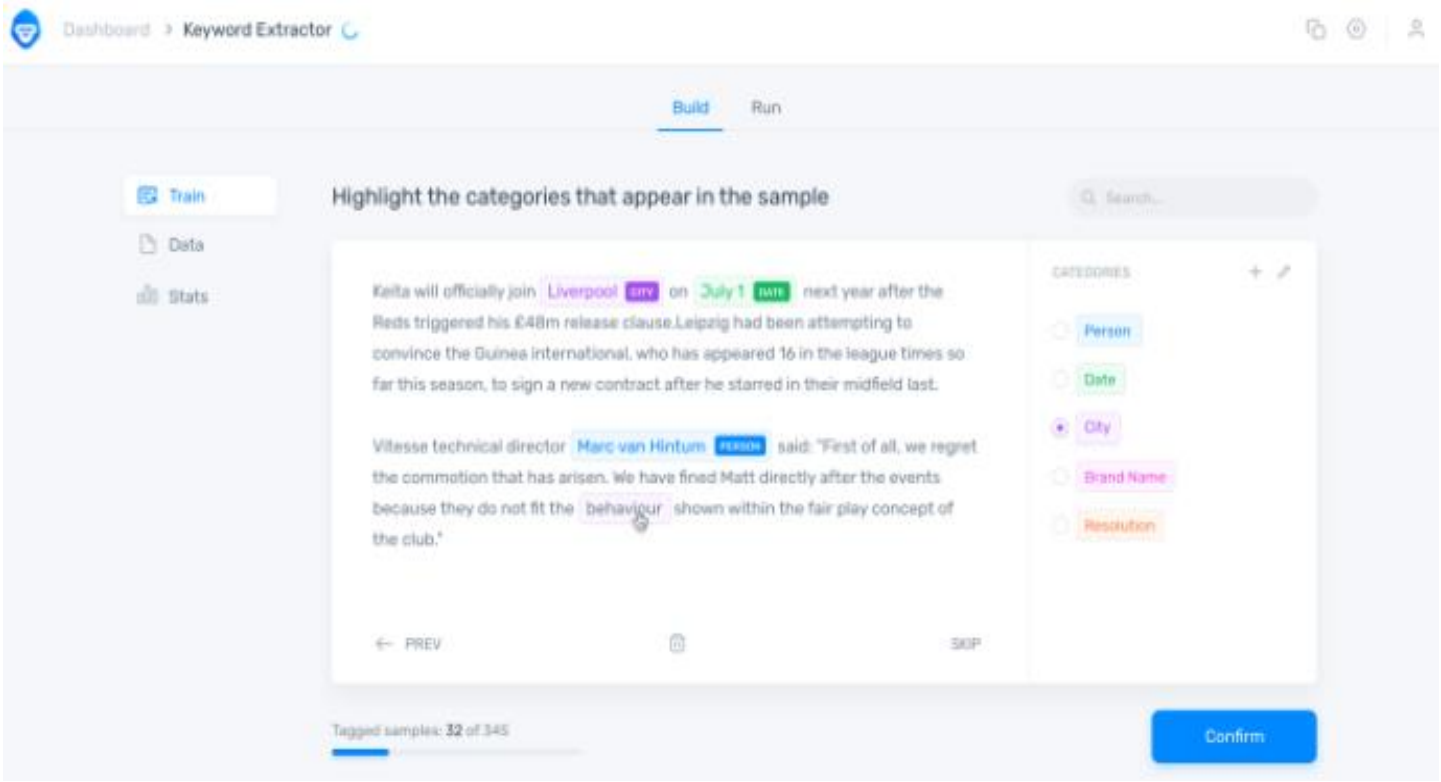
MonkeyLearn

Outline

MonkeyLearn is a sentiment analysis service that provides analysis for a range of sources such as social media sites, emails, web pages, documents.

An interesting feature of MonkeyLearn is that it not only recognises sentiment of text but also the topic and intent of the text. This would be useful when handling customer support as it would allow the user to see more specifically what people are saying about the company. For example, the a business using MonkeyLearn could search for tweets that are flagged as talking about "billing", so that these customers can be addressed as quickly as possible.

Another key feature of MonkeyLearn's product is the ability for companies to train their own classification models to identify features in text that will be key to them. This does not require any coding from the user, making it an accessible option for all their users. The screenshot below shows the screen the user can use to create their own model. The user can highlight different elements that fall under a category that they have created - this is then used to train a model which can be used to identify these features. This would be useful for a user wanting much more specific analyses.



Source: MonkeyLearn https://monkeylearn.com/

Reviews

Pros	Cons	Website	Comments
It has great facilities for reducing our work load and make us feel comfortable while dealing with complex problems. It has great user interface. It also provides tutorials that turns out to be helpful when you get confused at some point.	You can make only certain number of queries each month as per your plan. But that is still not a limitation. In terms of implementation, there is no such thing to dislike about Monkey Learn.	G2.com	Although I will try to make the software as easy to use as possible, I could include a brief tutorial for any users who are unsure about how to go about using it. The software I am aiming to create won't have limit on the amount of analyses the user can make.
The ease of use and setup. You don't have to be a programmer to set it and even use it. Their online dashboard and integrations	The fact that there are only 4 integrations!	G2.com	Automation looks to be an important for sentiment analysis so I will try to make the software as effortless to use as possible. To do this, I could use a dashboard as mentioned in this review which

are so easy, its just plug and play.			contains the bulk of the analyses so the user can easily access them. This reviewer has talked about the need for more integrations. This is something I should consider although Twitter will be the focus.

What I can apply to my solution?

A notable aspect of MonkeyLearn's approach to Sentiment Analysis is how they have broadened the analysis from Twitter to other crucial aspects of data analysis like the ability to automatically look through emails and documents. Expanding the scope of my solution to beyond Twitter could be a potential improvement later in development which would help deal with more formally written issues addressed to the companies such as emails written by customers. Another useful feature is the training of unique models. This should be a major consideration for the final solution despite its apparent complexity.

Microsoft Azure

Outline

Microsoft Azure is a cloud computing platform that provides tools for its users to aid with business development. The solution's primary function is not sentiment analysis but instead to provide number of tools including sentiment analysis as well as virtual machines, SQL database access, cloud storage etc.

Broad entity extraction

Identify important concepts in text, including key phrases and named entities, such as people, places and organisations.

Powerful sentiment analysis

Examine what customers are saying about your brand, and detect sentiment around specific topics.

Robust language detection

Evaluate text input in a wide range of languages.

Flexible deployment

Run Text Analytics anywhere – in the cloud, on-premises or at the edge in containers.

Source: https://azure.microsoft.com/en-gb/services/cognitive-services/text-analytics/

Azure's sentiment analysis tool has several features. One of these features is that it will understand and be able to perform sentiment analysis on a vast number of different languages. When you ask it to analyse a piece of text, it provides a percentage certainty of what language the text is written in e.g. language: Spanish, 100% and will then perform sentiment analysis. Azure also provides in-depth feature extraction which will pick out noticeable features of the text such as words that are indicative of certain sentiment. The most notable feature of their tool is a breakdown of the sentiment of each sentence in the text, helping deduce.

Reviews

Pros	Cons	Website	Comments
"The Azure Text Analytics is very easy to use. Go into the browser. Paste in your text and immediately pull keywords etc.**"**	"I have found the sentiment meter to be flawed and completely incorrect at times. The AI needs a little work interpreting context created by other sentences."	G2.com	This shows Azure is used more for general use rather than specifically social media. This review also highlights the inaccuracies of the model, showing the importance of accurate predictions.
"It's easy to use interface and the ability to handle with ease."	"There isn't a mobile or app version to use on the go or on smartphones or tablets"	G2.com	This how users value a good user interface. This should be a priority for development. Despite its potential use, I won't be making a mobile version of the software.
"Now we know what our customers are really thinking. This cloud based service takes raw text and can analyse customers sentiments and also capture key phrases, helpful for our marketing."	"Occasionally, it is not truly intelligent with slang, e.g "sick vibe" is a compliment, but it can translate as a negative"	G2.com	Again, the negative aspect of the review is concerned with the accuracy of the sentiment predictions. It is difficult to combat this issue as it is more of technological issue than a conceptual issue.

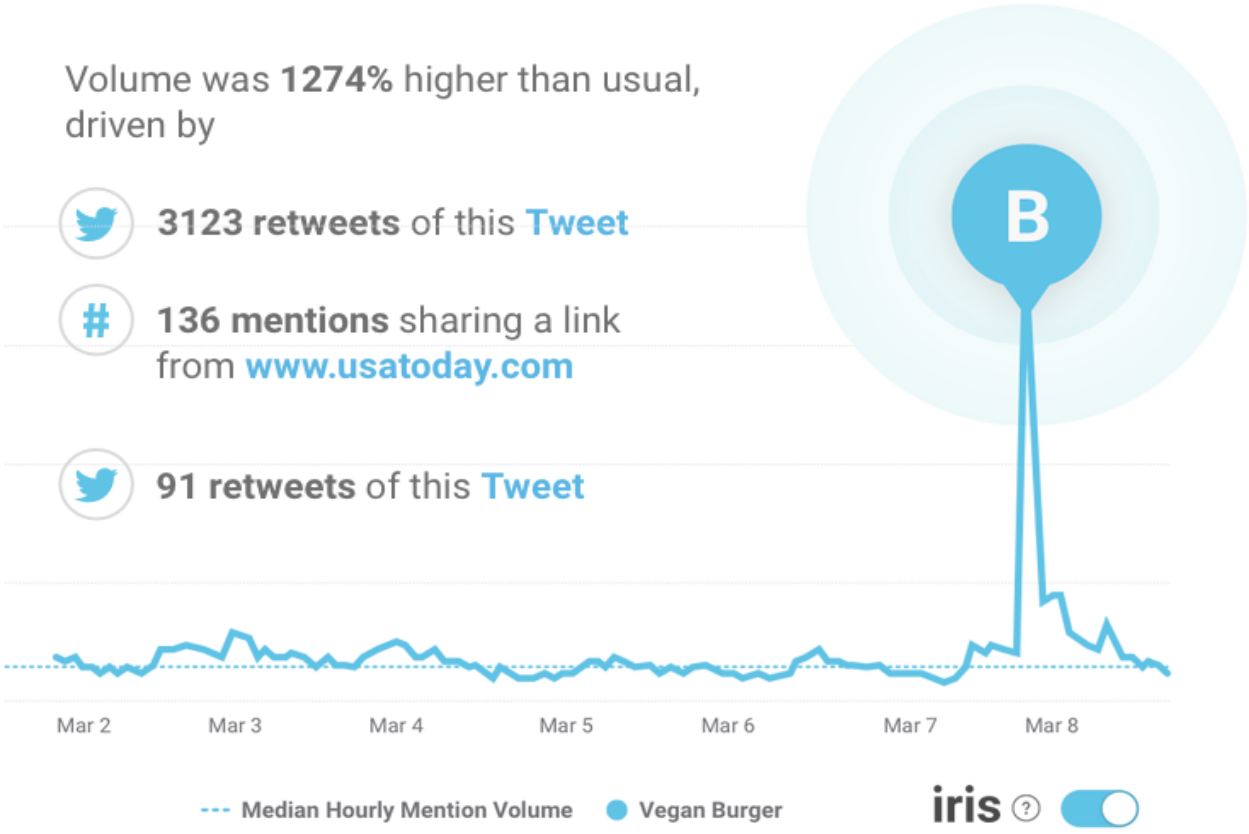
What I can apply to my solution?

The most significant feature of the tool is the highlighting of key features in the text. In the context of twitter sentiment analysis, I could implement a similar feature so that the companies are aware of common features of tweets such as common sources of criticism or key words found in tweets. A specific example of this would be my client using the software and it identifies that many people are tweeting about the poor availability for one of their products.

Brandwatch

Outline

Brandwatch is a sentiment analysis tool specialising in crisis management, brand management, competitor management, competitor analysis, customer experience, content strategy and influencer marketing. Brandwatch has access to a large pool of posts to aid these tasks, accessing over 1.3 trillion individual posts from 100 million sources. This enables them to create incredibly in-depth analyse on any given topic. The screenshot below represents the kind of interface Brandwatch provides. It gives important details on the tweets about the company which may be useful to implement.



Source: https://www.brandwatch.com/use-cases/brand-management/

Reviews

Pros	Cons	Website	Comments

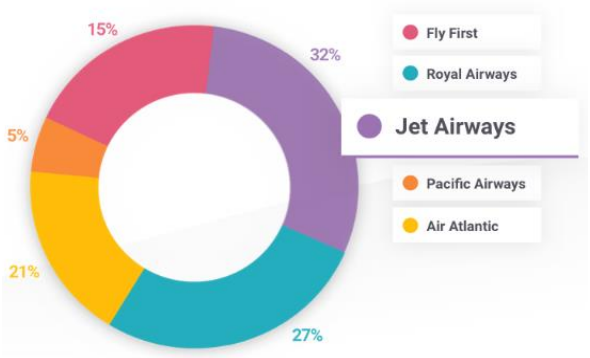
"Brandwatch has a well designed interface, it is very fast and almost totally customizable. Very useful tool that allows you to have an active presence on social networks. It has the ability to categorize negative mentions and keep them all in one place."	"The platform takes some time to learn and understand. the post list could be cleaned up a bit."	G2.com	Look at the positive part of the review, instead of just providing the user a list of tweets, it could be useful to categorize these tweets. The review also comments on the ease of use for the interface, something I am looking to prioritize.
"It is a great research tool for planning and near real-time analysis is crucial to see trends, helps to identify influencers to expand the reach of a campaign. Brandwatch is probably the best solution out there for your social listening needs, it has a beautiful and intuitive user interface. allows me to analyze and study conversations about the brand, find new ideas, ideas, needs, innovations or opportunities."	"Simple details, they should improve searchability not to mention those repeating comments."	G2.com	The suggested improvement here shows that the user may want to have specific criteria when looking for tweets about their company I could implement this as a series of drop down lists which the user could use to select criteria.
"I like the simplicity with which the data is displayed and the ease and help in creating queries."	"I don't like that a dashboard cannot be saved automatically"	G2.com	I could implement the ability to save and print out dashboards.

What can I apply to my solution?

From looking at the reviews, the users highly value the simplicity and ease of use of the software given the user interface. It is important that this becomes a priority for the design of the solution as without ease of use and an easy to navigate user interface, the user may lose incentive to use the software entirely. I could use visual elements similar to those shown below. The user may benefit from comparison with their direct competitors so this should be something to take into consideration. I should also allow the user to save the dashboard and be able to print it out.

The most common negative, neutral,

The company's online presence



Source: https://www.brandwatch.com/use-cases/brand-

Takeaways from research of existing solutions

Looking at the reviews, there is an evident need for a clear and an easy to navigate user interface. This must be an important consideration for designing the software as this was a common comment for the positive aspects about the existing products.

Some of the existing solutions offer the creation of machine learning models by the user. This could be useful for highlighting specific data points for the user however, this may add unnecessary complexity for the user and the need for the system to be easy to use for my client is apparent from the conducted interview. As a result, this feature is not a priority for creation of the solution.

Something I am likely to implement is the tool that highlights prevelant features of the text i.e words or phrases that appear often in the text. This may help provide more specific insight into what people are talking about. This could be implemented easily with a word cloud similar to the example shown in the research for Brandwatch.

It seems as though these existing solutions are able to provide analysis of multiple different sources rather than a single site like Twitter. Given the time it would take to implement the analysis of multiple sites. For now, the solution will solely focus on Twitter as a source for data. The analysis of other sites should be a consideration for future development.

Key Features of the proposed computational solution

Following research of what is required from the system through the client interview and research of existing solutions, I have created the following list of features necessary to the solution's success.

- **User login/ register details linked to a database**

The user will be able to login to the software so that the system knows which data it needs to be showing to the user. The user will also be able to change their password once logged in. This login system will be built on top of a database.

- **Main dashboard displaying analysis on what people are saying about the user's company**

The software will provide a dashboard which will be the primary source of information for the user. This will contain the most common opinions of the user's company; the company's performance against their competitors; The most recent tweets about the company; the number of positive, negative, neutral tweets etc. This information will be automatically collected by the system without user intervention and will be displayed when the software is opened. This main dashboard will also be able to display results graphically to make data easier to interpret.

- **Windows showing analysis of tweets based on topic provided by user**

The user will be able to enter a topic they want to find twitter users opinions on and then the program will automatically find tweets that fall under this topic. The user will also be able to select the number of tweets they want the system to search for and analyse. The system then will be able

find tweets that meet these criteria created by the user, perform analysis on them and then display the results graphically. The topics to be analysed can be anything the user may find helpful.

The user will be able to make as many of these requests as they want so they can build up a collection of the public's opinion on various topics. This is in addition to the provided sentiment analysis on the company which is shown on the main dashboard.

- **Machine learning model that performs sentiment analysis**

The system will be using a machine learning algorithm which will be able to read text and predict the text's sentiment i.e how positive or negative it is. This model will be used to perform analysis of the necessary tweets to provide the data described in the features above.

- **Storing of sentiment of the company and the user topics in the database**

The system will be able to connect to the database and store the results of any analyses results when analysing the tweets about the company or the user topics. The system will then use this data when the software is used at a later date.

- **Logging data over time**

Any necessary analyses will be logged in the database so the user can be informed about the changes in people's opinions over time. The system will be automated, so the user won't need to be actively using the software for this logging to happen.

System Requirements

Hardware

Minimum:

- *Intel® Core™ i3-1005G1 Processor CPU*
  The model used by the software will be making predictions on sometimes large quantities of text so the CPU required will need to be able to handle this. This i3 should be able to handle this.

- *A consistent internet connection (10+Mbps)*
  The system will be scraping tweets from online. A consistent internet connection means that if the system is left running it can continue to scrape tweets. Without an internet connection, retrieval of tweets from online becomes impossible so an internet connection is an absolute minimum requirement for the system.

- *External peripherals, e.g keyboard, mouse, monitor*
  The user needs to be able to provide basic input for the system and needs to be able to view the user interface of the system.

Recommended:

- *Intel Core i5-4460 Processor CPU*
  An i5 will also be able to handle the analyses of the text, but will just make the experience smoother if large amounts of text are to analysed.

- *M.2 SSD*

When the user initially logs in to the system, the system will need to load the model it will be using to perform the sentiment analysis. Often, these files containing these models are quite large in size and so can be slow to load in to memory so the use of a comparably fast SSD should mitigate this issue.

- *Strong internet connection and high bandwidth (30+Mbps)*
  A strong internet connection means the system will be able to be able to consistently scrape tweets in the background. A quality internet connection also means that if the program is running and repeatedly pulling data from the internet, it will not be using too much bandwidth and so shouldn't slow other machines on the network.

Software

Minimum:

- *Python Libraries*

The program will be running from Python3.7.0 and will need to use the following libraries that are not preinstalled with Python:

  - **NumPy, Pandas, Matplotlib:** These libraries are used when working numerical data. Matplotlib is used for creating graphs using data represented using data structures provided by NumPy and Pandas.
  - **Tweepy:** this library is used for retrieving Twitter tweets from online and Re (Regular expression) is used for cleaning (formatting) the tweets.
  - **NLTK (Natural Language Toolkit):** NLTK provides a series of tools for working with Natural Language Processing. This includes the type of model I will be training as well as some tools necessary for pre-processing.
  - **PyQt5:** PyQt5 is used to create the graphical user interface that these libraries will use to display information
  - **Cryptography:** This library will be used when encrypting passwords to be saved to the database.

- *Windows, MacOS or Linux OS operating systems*

Python applications can be used on all three of these operating systems. I have not included mobile operating systems as the solution is not intended to be used on mobiles.

Recommended:

- *Windows 10 Operating System*

The system is to be created and tested on a machine using Windows 10 so it is preferrable but certainly not essential for the user to also be using Windows 10.

Potential Limitations of the solution

The most prominent limitation of the solution is an inaccurate model. Without an accurate enough model, the software becomes completely inadequate. An accurate model means that user has a good gauge of people's feelings. If these predictions are inaccurate too often i.e the model has below 75% accuracy, the user may make misinformed decisions.

A more minor limitation of the system is the potential lack of specificity that will be provided for the user - typically in sentiment analysis, text is described as "positive", "neutral" or "negative" which may not provide the full picture. However, this is more of an inherent issue with the concept of sentiment analysis rather than the solution I will be creating.

An potential limitation of the software is the boundaries of using the '*Standard'* (free) access to the Twitter API. The Twitter API is what I will need to use to access the tweets and so only using the free version could become problematic as this version limits the tweets scraping to 18,000 tweets per a 15-minute window. This will be an issue if there are many users of the system who are all scraping tweets at the same time which means the system could reach the limit and so some users may not be able to retrieve the necessary tweets. However, this this should not be a significant issue as it is highly unlikely the system will reach that many tweets to scrape in such a short amount of time.

Success Criteria

The following is a list of criteria based on all the previous research and consideration completed thus far.

Aesthetics

Requirement	How is it measurable?	Is it essential?
The software will have a professional look with consistencies in font style and font size large enough to be easily readable	A screenshot of the window while the program is running will be provided	✓
The software needs to have a simplistic design so that it is not difficult to use.	A screenshot of the main window with large readable elements is to be provided.	✓
The windows for the user interface will be large enough to contain any visual elements that the user would need to see.	A screenshot of the dashboard with all relevant data about the company is to be provided as well as a screenshot of a window showing analyses of a user-requested topic.	✓
The user will be able to change the colour scheme of the user interface to match their preferences.	Two screenshots of the window will be provided, each with different colour schemes.	✗

Inputs

Requirement	How is it measurable?	Is it essential?
User authentication in the form of user login and password. This will be validated in a database.	Screenshots will be provided showing what happens if the user gets the password incorrect will be provided as well as what happens when the password is correct.	✓
Users will be able to create a new account, entering login details and the company they belong to.	A screenshot of the signing up window will be provided.	✓
The user will be able to input a topic that they want to find the public's opinions on.	A screenshot of the screen where the user can enter their own topic will be provided.	✓
The user will have the option to view sentiment analysis requests made in the past or make a new request about a new topic.	A screenshot of a window showing the analysis of a user-requested topic.	✓
The user will be able to select the option to change the password to their account.	A screenshot of the "change password" screen will be provided.	✓
Users can create their own machine learning models to identify specific features in	A screenshot of the screen where the user can create their own models will be provided.	✗

tweets like whether the person is tweeting about the customer service or quality of the product, for example.		

Outputs

Requirements	How is it measurable?	Is it essential?
A login/sign up window is provided prompting the user to enter their details	A screenshot of the login/sign up window will be provided.	✓
The software will be displayed on a large main window so that data is easy to read	A screenshot of the main window will be provided.	✓
The main window has a dashboard containing analysis on the user's company	A screenshot of the dashboard will be provided.	✓
Where appropriate, data will be shown graphically.	A screenshot of the embedded graphs on the windows will be provided.	✓
Sentiment analysis data will be called from a database to show sentiment change over time.	A screenshot of a graph showing changes in trends over time will be provided.	✓
The user will be able to change the type of graph representing some of the on-screen analyses. E.g change a bar chart to a pie chart or vice versa.	Two screenshots will be provided, each with a different graph type representing the same data.	✓
There will be a settings menu where the user can adjust the software settings	A screenshot of the settings menu will be provided	✓
There will be a help screen where users can find out how to use the software and find out more about the software	A screenshot of the help screen will be shown	✓

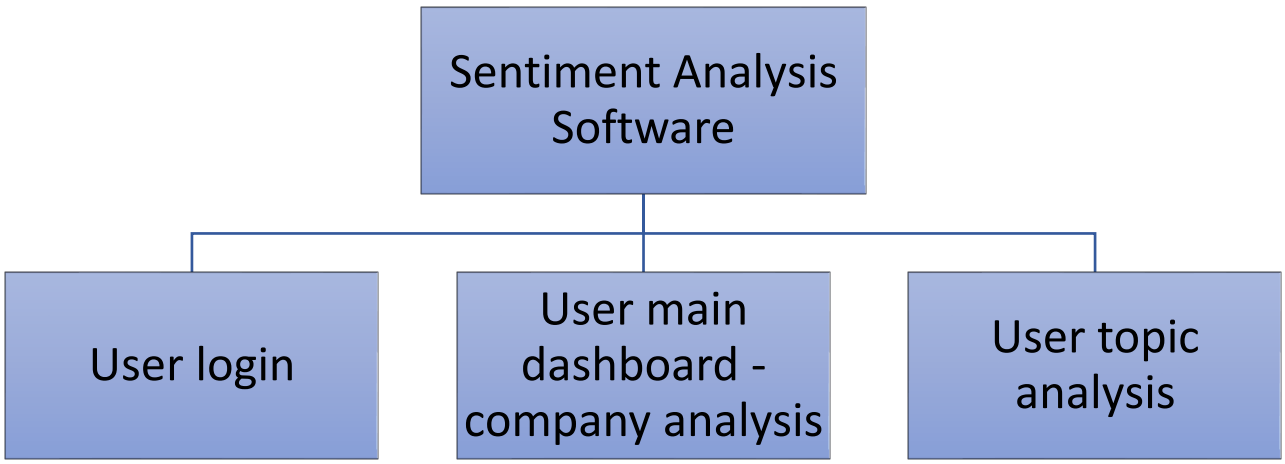
Design

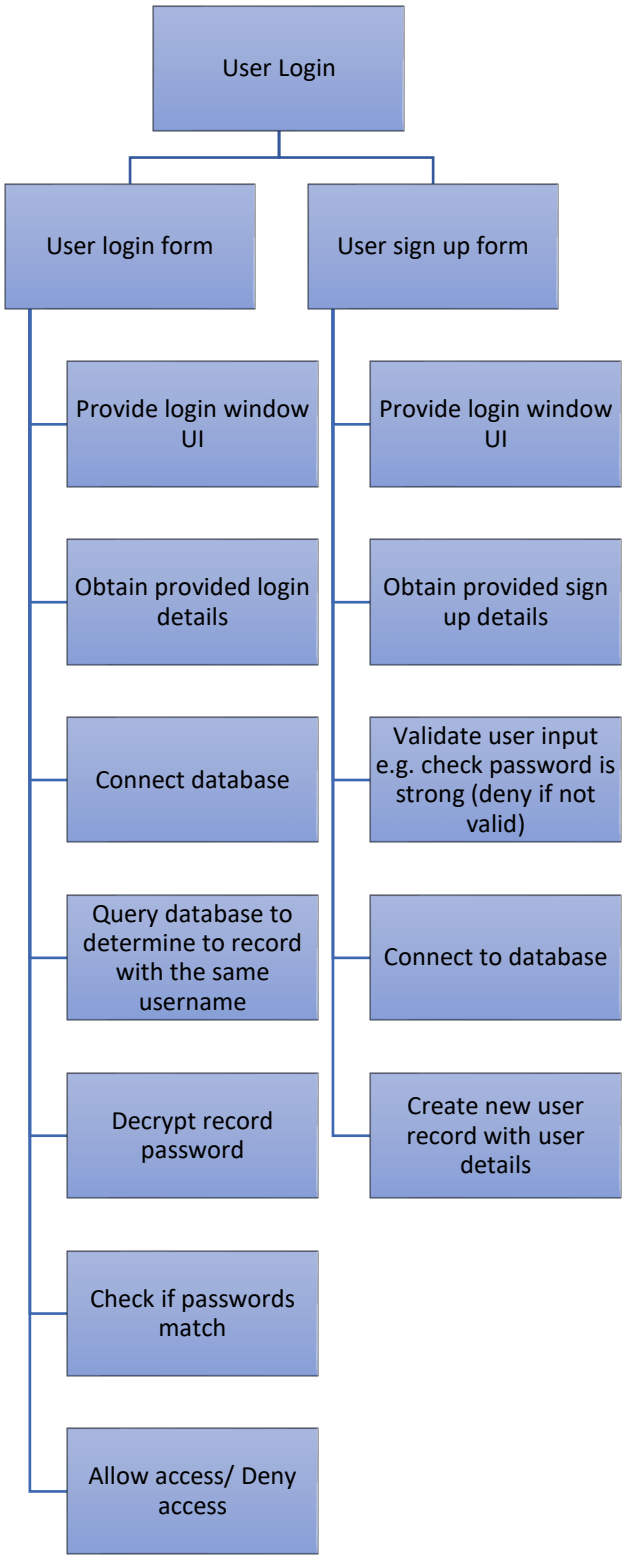
Decomposing the solution (Top-down designs)

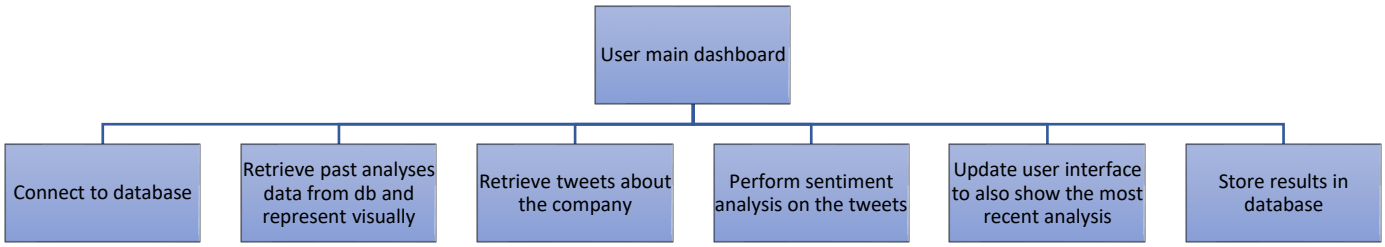
Justification of breaking down the problem

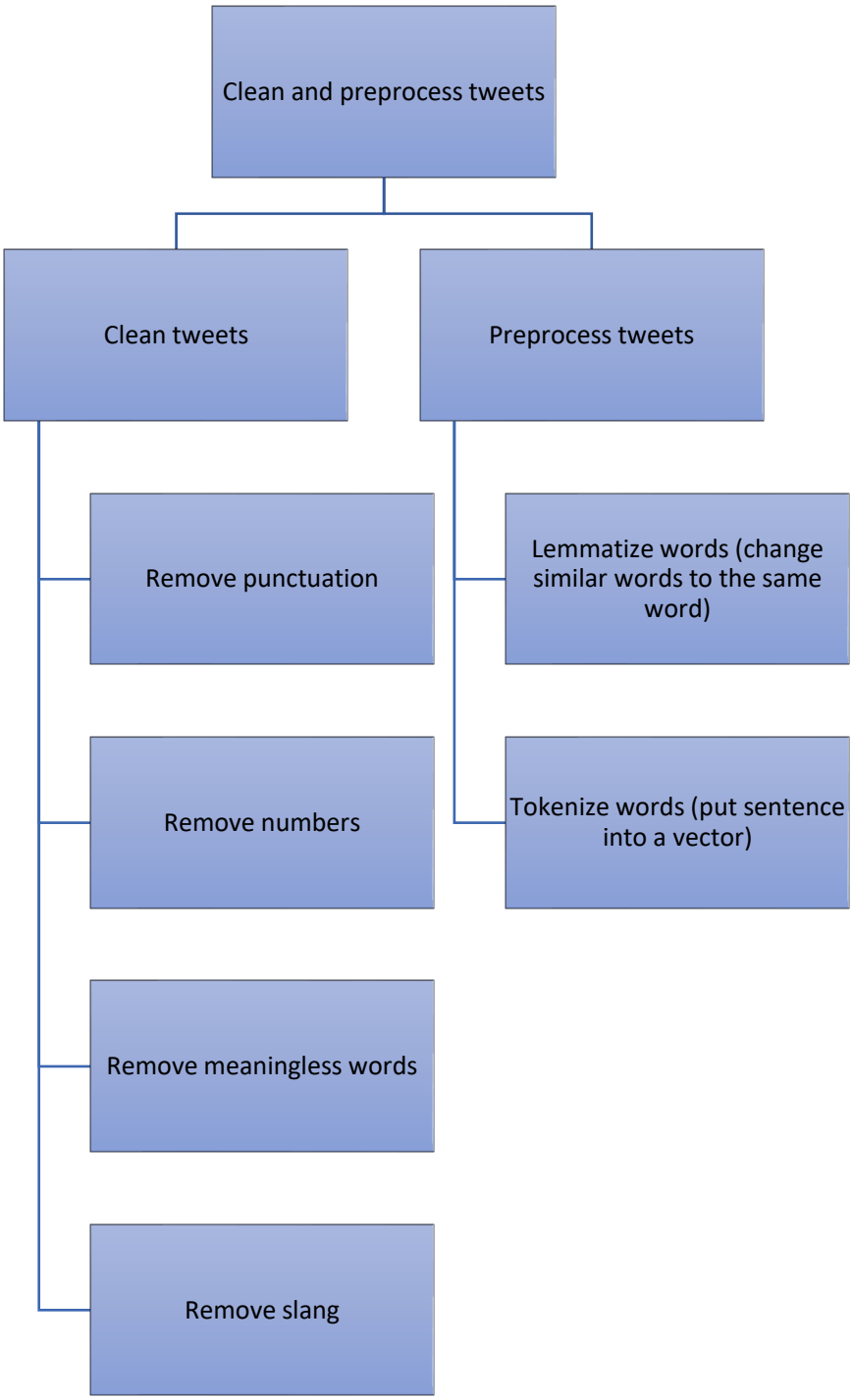
Conceptually, the proposed solution seems quite complex and by breaking down the problem into different parts, it means less time during development is spent trying to understand the logic required to create the primary utilities and how they fit into the broader application. An example of one such utility is the system's need to retrieve tweets from online. During decomposition, we can consider the required features of the tweet scraper that retrieves the tweets and break down these to their most basic components like taking user input and then connecting to the Twitter API and making a request to Twitter for the tweets. The simpler components can then be developed and tested one by one eventually forming a complete solution. The following are hierarchal diagrams (top-down designs) demonstrate decomposition of the solution, breaking down the solution's primary high-level functionality into easier to tackle concepts.

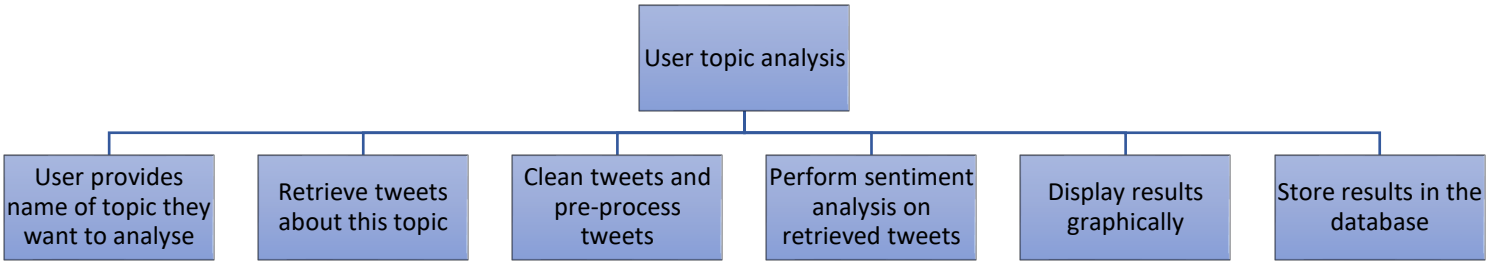
Primary capabilities of the solution



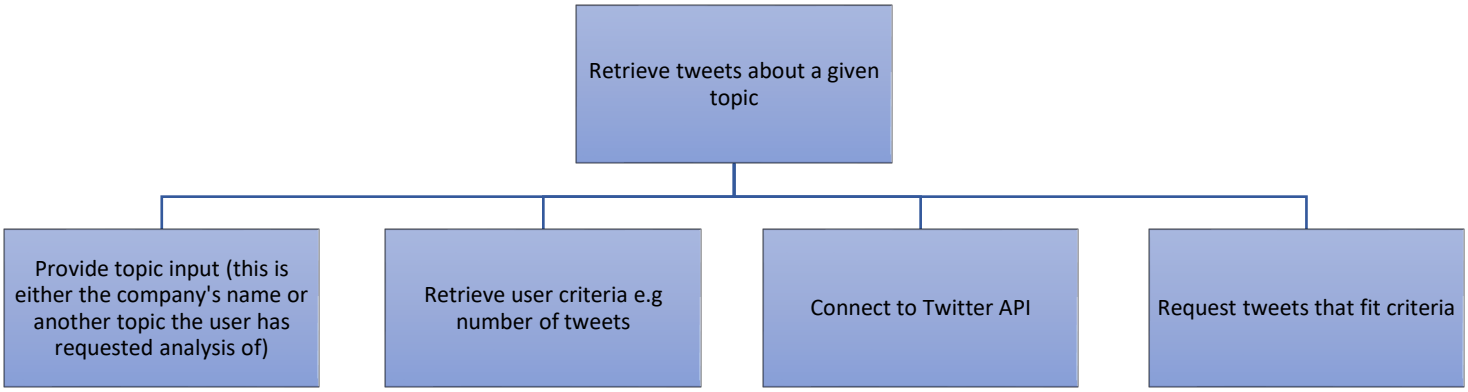

User Login
User login form
User sign up form
Provide login window UI
Obtain provided login details
Connect database
Query database to determine to record with the same username
Decrypt record password
Check if passwords match
Allow access/ Deny access
Provide login window UI
Obtain provided sign up details
Validate user input e.g. check password is strong (deny if not valid)
Connect to database
Create new user record with user details

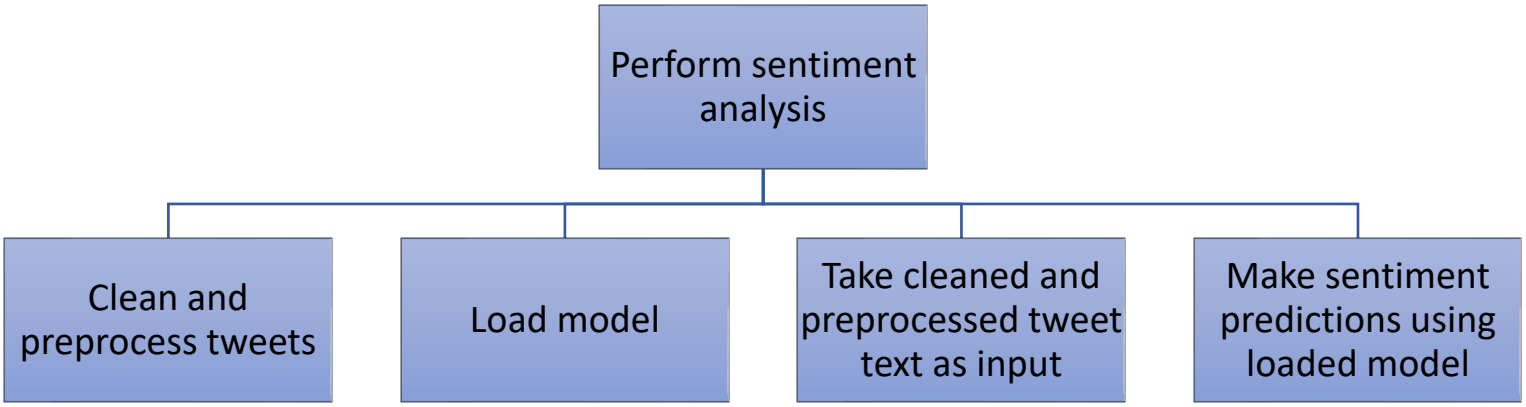

User main dashboard
Connect to database
Retrieve past analyses data from db and represent visually
Retrieve tweets about the company
Perform sentiment analysis on the tweets
Update user interface to also show the most recent analysis
Store results in database


Clean and preprocess tweets
Clean tweets
Preprocess tweets
Remove punctuation
Remove numbers
Remove meaningless words
Remove slang
Lemmatize words (change similar words to the same word)
Tokenize words (put sentence into a vector)



Secondary Features





Database Design

For the solution to perform as intended, it will require the use of a database to access data. This is primarily because of the need to present changes over time.

Users table

This table is required for the user login feature of the solution. The solution will be able to query this database for validation of user login details. There are 4 user access levels that I have explained in greater detail later in the report. These are important as I do not want standard users of the solution to be able to make changes to the database.

Name	PK/FK	Data type	Null?
User_ID	PK and FK	Int(Autoincrement)	No
User_username		String (MAX)	No
User_password		String (MAX)	No

User Topic table

This is a joining table for the Users table and the Topics table. This is necessary as it is a many-to-many relationship between these two other tables so intermediary is necessary for one

Name	PK/FK	Data type	Null?
User_ID	PK	Int(Autoincrement)	No
Topic_ID	FK	String	No

Topic table

The solution will be able to store sentiment analysis results on various topics. This table is for storing information on these topics. Each record will contain a topic name describing what this record is referring to i.e the company's name. Storing the tweets with information like their author, date they were posted, and their predicted sentiment means that this information can be accessed for visualisation again after the software is closed. Storing the dates and sentiment of tweets specifically means that graphs can be created showing the changes in sentiment over time. This will be useful for the user trying to see how different events have affected the opinions of consumers on the company and on the topics the user has requested.

Name	PK/FK	Data type	Null?
Topic_ID	PK	Int(Autoincrement)	No
Topic_name		String	No
Topic_sentiment		Float	No
Topic_datetime		Datetime	No

User Companies Table

This is the table that links the different users of the system with the companies that these users belong to. Again, this is necessary as users and companies have a many-to-many relationship.

Company Table

This is the table where the opinions of the company itself are stored.

Name	PK/FK	Data type	Null?
Company_ID	PK	Int(Autoincrement)	No
Company_name			

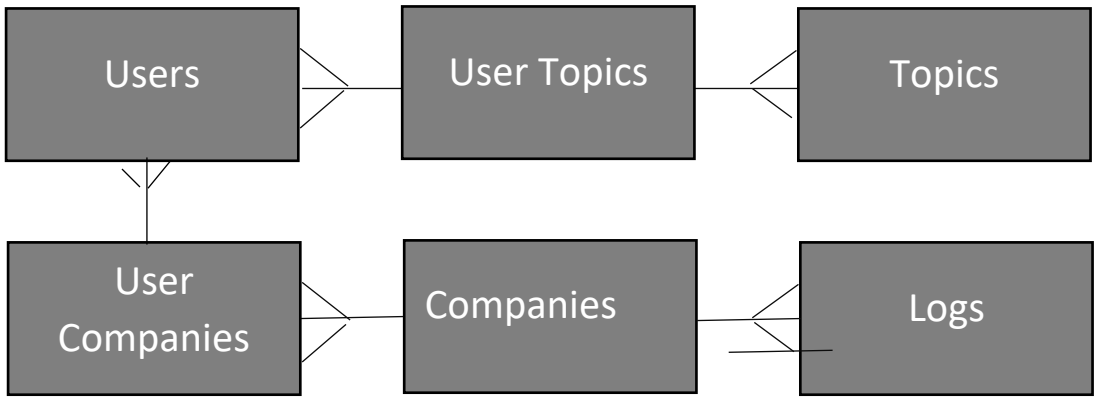
Logs table

The log table will be used to store any changes in sentiment data that occur over time. It is necessary to hold this in its own table to reduce data redundancy.

Name	PK/FK	Data type	Null?
Log_ID	PK	Int(Autoincrement)	No
Log_sentiment		Float	No
Log_number_of_tweets		Int	No
Log_date		DateTime	No

Entity relationships

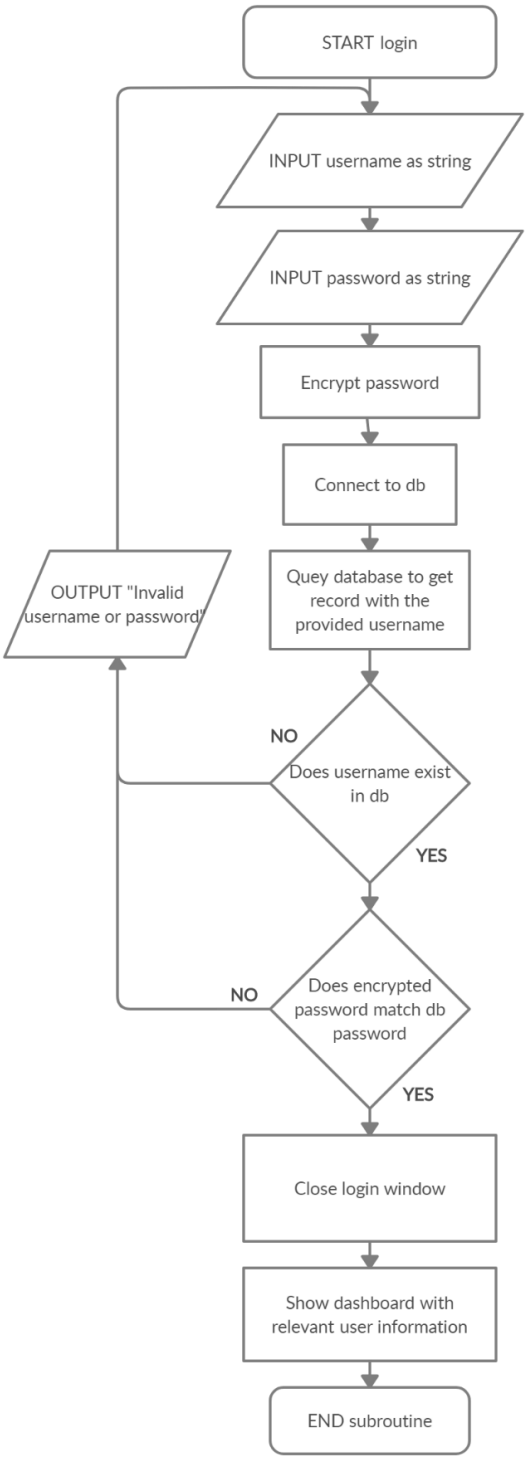
The following are entity relationships representing the relationships between various tables. At a high level is a many-to-many relationship between Users and Topics and between Companies and Logs.



Algorithms - Subroutines

Login

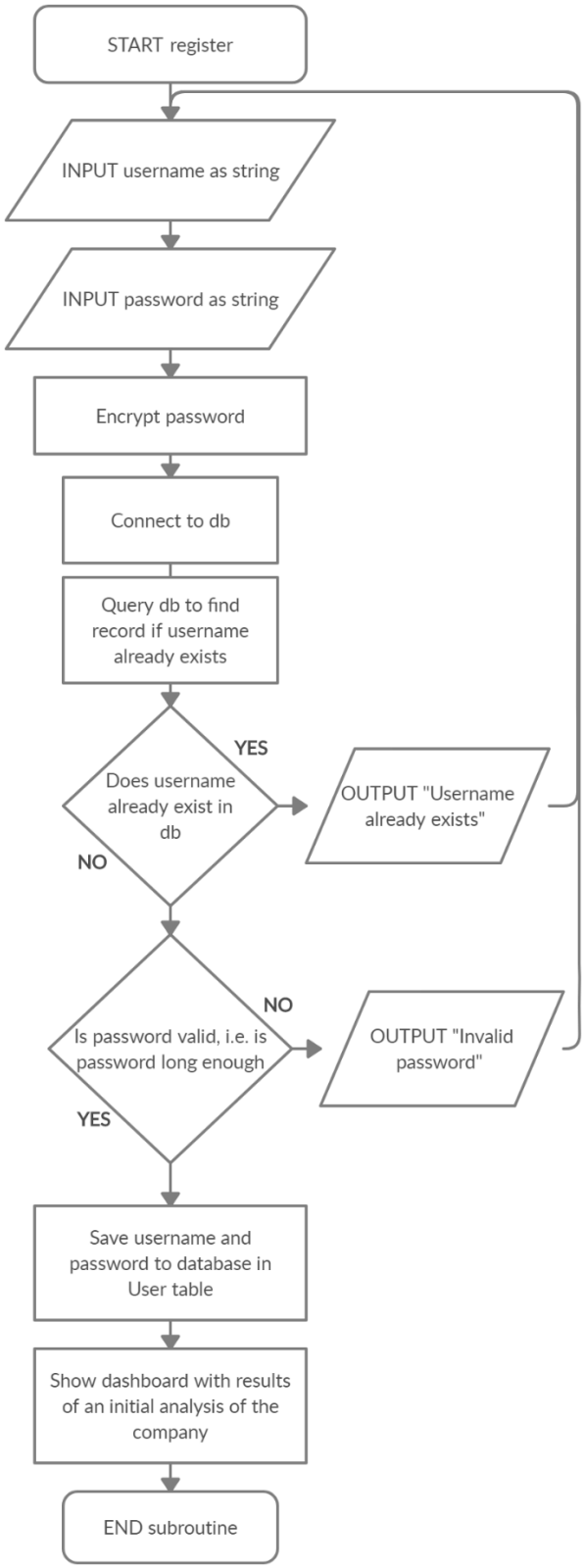
This form is for users to login and register into the system. The login function will take a username and password from the user. It will then connect to the database and verify whether these login details are correct. If so, the login form will close and then the dashboard window will open. This dashboard will then display all information that is relevant to the user.



```
function login():
        // Get entered username
        string username = input()

        // Get entered password
        string password = input()

        // Encrypt the password
        string encryptedPassword = encrypt(password)

        // Connect to database and get the user record
        database.connect()
        record userRecord = database.query("SELECT User_password FROM Users WHERE User_username=username")
        database.disconnect()

        // Check if username exists in database
        if userRecord == False then
                output("Username does not exist")
                return False
        endif

        // Check if password is valid
        if encryptedPassword == userRecord.User_password then
                ouput("Valid login details")
                return True
        else
                output("Invalid username or password")
                return False
        endif
endfunction
```

Register

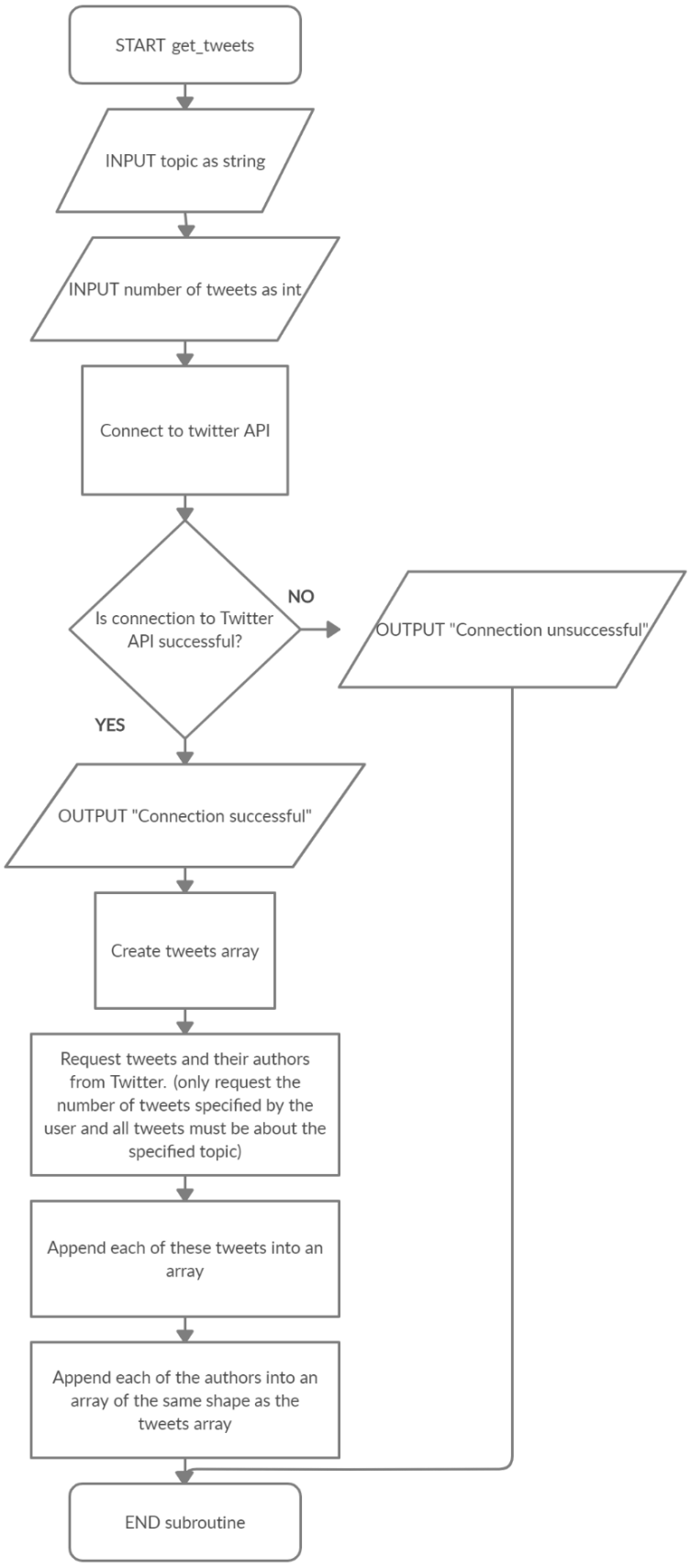
This form involves the user being able to register an account for the software. For this to work It needs to ensure the user details are valid such as the account name not already existing and the password is secure. It then needs to save the user details to the database.


START register
INPUT username as string
INPUT password as string
Encrypt password
Connect to db
Query db to find record if username already exists
Does username already exist in db
YES
NO
OUTPUT "Username already exists"
Is password valid, i.e. is password long enough
NO
YES
OUTPUT "Invalid password"
Save username and password to database in User table
Show dashboard with results of an initial analysis of the company
END subroutine

```
function register():
        // Get entered username
        string username = input()

        // Get entered password
        string password = input()

        // Get entered company
        string company = input()

        // Connect to database and get the user record
        database.connect()
        record userRecord = database.query("SELECT User_password FROM Users WHERE User_username=username")
        database.disconnect()

        // Check if username exists in database
        if userRecord == True:
                output("Username already exists")
                return False
        endif

        // Check if password is valid
        if password.length() > 7 then
                database.connect()
                database.query("INSERT INTO Users (user_ID, username, password,company) VALUES (NULL, username, password, company)")
                database.disconnect()
                return True
        else
                output("Invalid password")
                return False
        endif
endfunction
```

Retrieving Tweets

This algorithm involves scraping the tweets from online so that they can be processed. To do this, the name of the topic the user wants to analyse is required. The number of tweets requested is also required as the system needs to consider the limits to the number of tweets that can be scraped from.

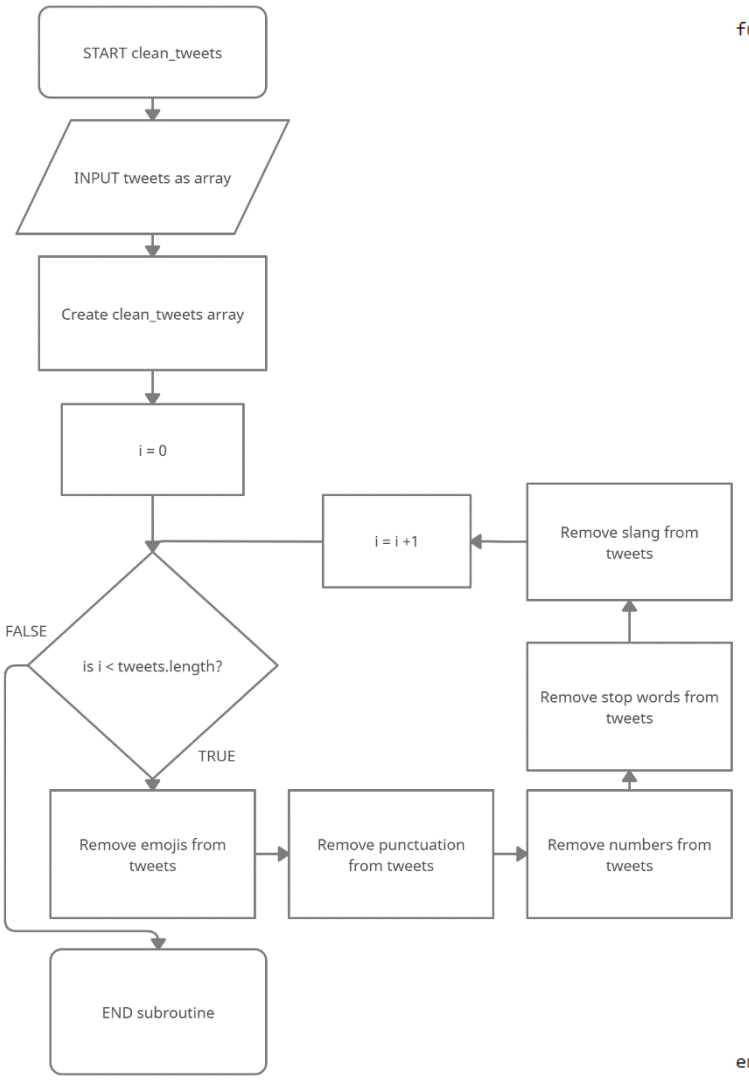

START get_tweets
INPUT topic as string
INPUT number of tweets as int
Connect to twitter API
Is connection to Twitter API successful?
NO
OUTPUT "Connection unsuccessful"
YES
OUTPUT "Connection successful"
Create tweets array
Request tweets and their authors from Twitter. (only request the number of tweets specified by the user and all tweets must be about the specified topic)
Append each of these tweets into an array
Append each of the authors into an array of the same shape as the tweets array
END subroutine

Cleaning Tweets

A crucial algorithm needed for the software is one that will be able to take a series of tweets as input and then clean them, removing any unreadable elements so that they can be processed by the model. Cleaning of the tweets will involve identifying and removing specific elements of the text that have no meaning for the computer. The cleaning process will involve the removal of:

- Emojis
- Punctuation
- Numbers
- Stop words (words like “a” or “the” which add no meaning to the text)
- Colloquial language (slang)

The following pseudocode is an abstraction of how the program will remove these components. The main program will use Python Regular Expressions to make text cleaning more efficient. For emojis, as the computer cannot interpret the emoji icons themselves, they each have a unique Unicode id which has been implied using the “emojiCodes” array (there are far more than 2 Unicode strings representing emojis but for the sake of space I have only included 2 in the pseudocode).



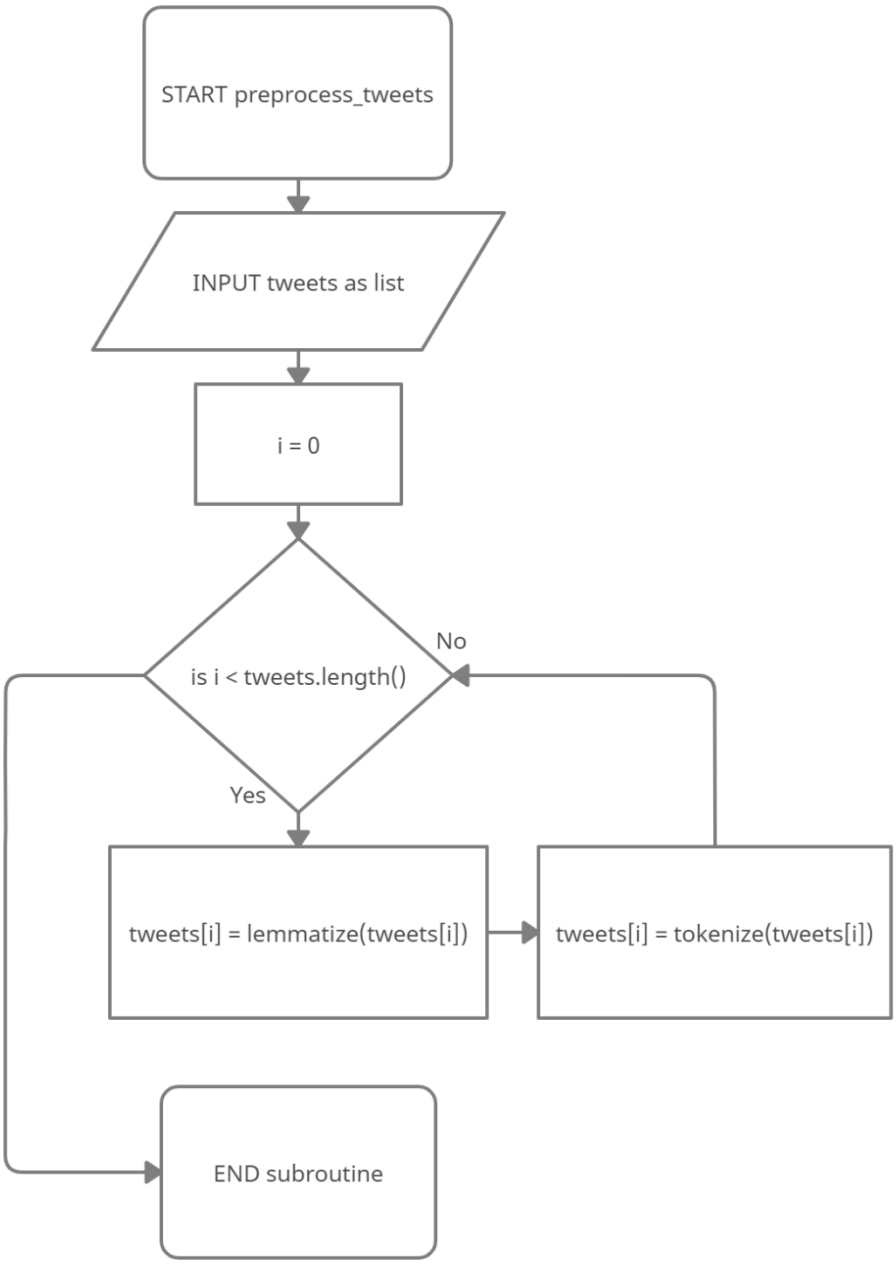

```
function clean_tweets(tweets): //Take tweets array as parameter

        // Declare an array for the cleaned tweets
        array cleanTweets = []

        // Declare an array of emoji Unicode
        array emojiCodes = ("U+1F600",...,"U+1E007F")

        // Declare an array of slang
        slangWords = ['pls','plz',...,'thx']

        // Declare an array of stop words
        stopWords = ['the','a',...,'if']

        // Iterate over each unclean tweet
        for i=0 to tweets.length-1:
                tweet = tweets[i] // Current tweet being cleaned

                // To remove emojis
                for j=0 to emojiCodes.length-1: // Iterate over emojis
                        emojiCode = emojiCodes[j]
                        tweet = tweet.remove(emojiCode)
                next i

                // To remove slang
                for j=0 to slangWords.length-1: // Iterate over slang words
                        slangWord = slangWords[j]
                        tweet = tweet.remove(slangWord)
                next j

                // To remove stop words
                for j=0 to stopWords.length-1: // Iterate over stop words
                        stopWord = stopWords[j]
                        tweet = tweet.remove(stopWord)
                next j

                // Append clean tweet to cleanTweets
                cleanTweets.append(tweet)
        next i

        // Return the cleaned tweets
        return cleanTweets

endfunction
```

Tweet Pre-processing

This algorithm follows the tweet cleaning algorithm to get the tweets ready for the model. This processing part involves two process: lemmatization and tokenization.

Lemmatization involves reducing similar words to a single word to reduce the overall complexity of the text. An example of this is reducing “changed”, “changing”, “change” and “changes” to “change”. This means the model learns the meaning of less words and is able to better understand the meaning of the word that the others have been reduced to.

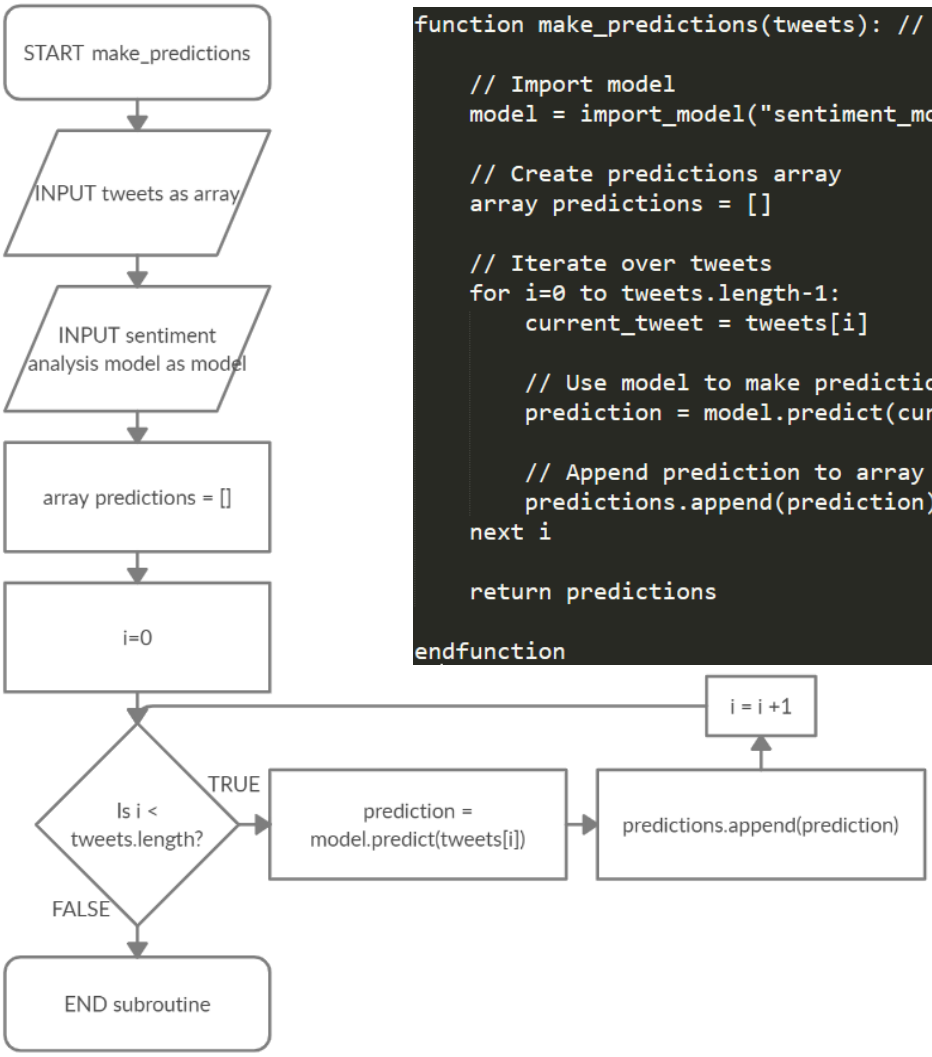
Tokenization involves breaking the text down into individual words. An example of this is breaking down the sentence “I love pizza” into [“I”, “love”, “pizza”].



```
function preprocess_tweets(tweets): // Take cleaned tweets as parameter

        // Iterate through the tweets
        for i=0 to tweets.length-1:

                // Lemmatize the current tweet
                tweets[i] = lemmatize(tweets[i])

                // Tokenize the current tweet
                tweets[i] = tokenize(tweets[i])

        next i

        // Return the pre-processed tweets
        return tweets

endfunction
```

Performing sentiment analysis

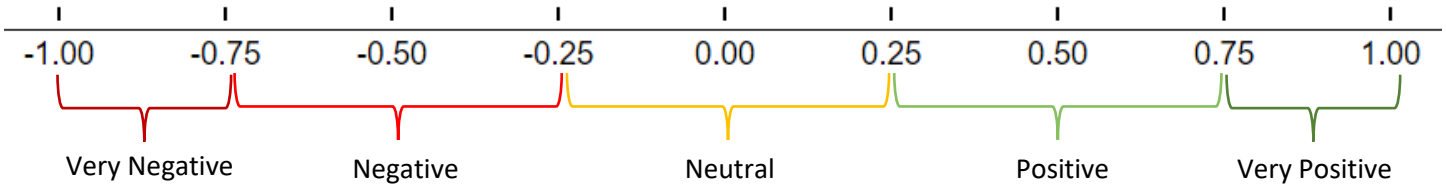
This algorithm is where we now use the model to make predictions on the tweets. This can simply be done by importing the machine learning model then iterating over the tweets and making predictions on each one and storing these predictions in an array.




```
function make_predictions(tweets): // Take clean tweets as parameter

    // Import model
    model = import_model("sentiment_model")

    // Create predictions array
    array predictions = []

    // Iterate over tweets
    for i=0 to tweets.length-1:
        current_tweet = tweets[i]

        // Use model to make prediction
        prediction = model.predict(current_tweet)

        // Append prediction to array
        predictions.append(prediction)
    next i

    return predictions

endfunction
```

The prediction of a tweet's sentiment that the model will provide will initially be in a numerical form as oppose to "positive" or "negative". These values will be within a range that describes how positive or negative the tweet is. This range could be between -1 and 1 meaning that if the model outputs 1 for a tweet it will mean that tweet is positive and if the model outputs -1 it means the tweet is negative. Numbers between -1 and 1 indicating a tweet is not wholly positive or negative need to be dealt with accordingly with the system deciding whether they fall in to the "positive", "negative" or "neutral" category.

The diagram below shows an example of the range of outputs the model may produce and how this translates to its sentiment. (These values are entirely arbitrary and so may be different in the final program depending on the output of the model i.e the model outputs in a range 0 to 2 instead of -1 to 1).



The model's predictions of the "tweets" array will be stored in the "predictions" arrays. These arrays will have the same length so each tweet will have a corresponding prediction so that they can be accessed using the same index. I have demonstrated this in the diagram:

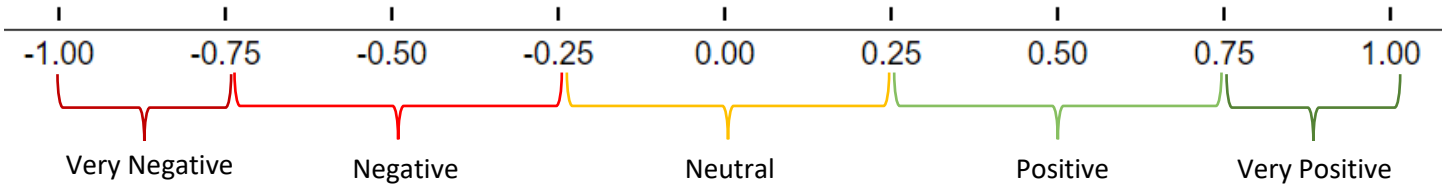
INDEX:	0	1	2	3
"tweets" array:	"I hate this product"	"I like this product	"The product is OK"	"I love this product"
"predictions" array:	-1.0	0.6	0.1	1.0

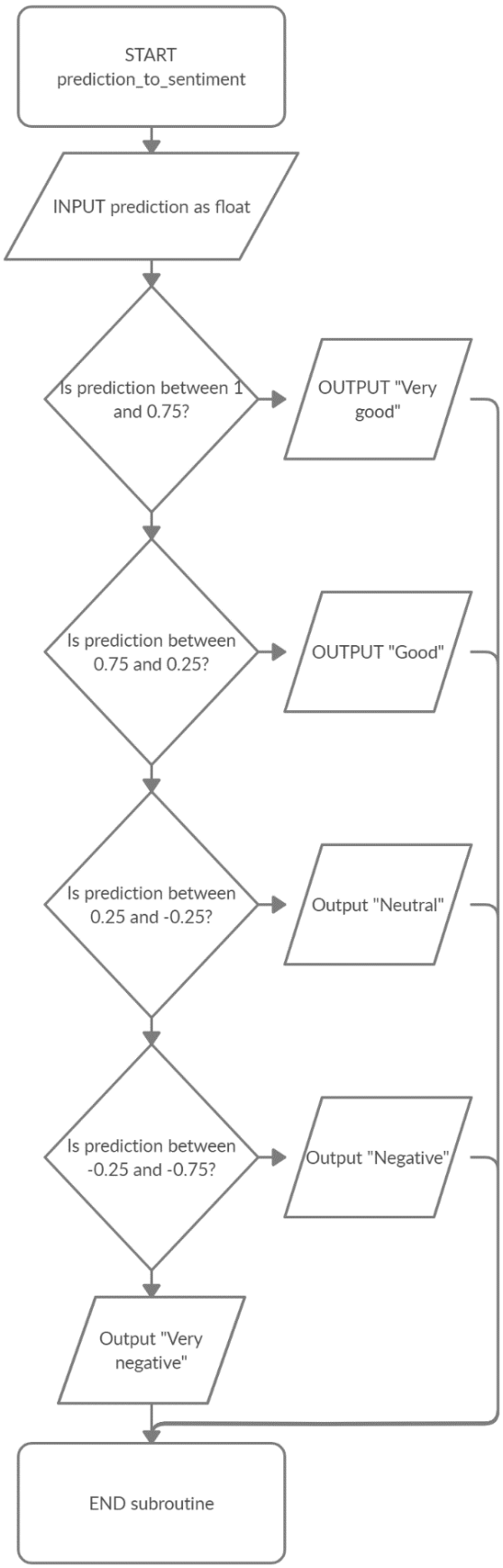
In practice, having the arrays the same length means that if I want to find the prediction for the nth tweet ("tweets[n]") I can just call "predictions[n]" where n is the index.

Converting predictions to sentiment

Now we have two arrays, the "tweets" array holding the tweets about the company/topic and the "predictions" array holding float values representing how positive or negative its corresponding tweet is. For the user, displaying an arbitrary number representing the sentiment of tweets will not be much use so these predictions need to be converted to tangible sentiment labels.

This can be done by looking using the prediction/ diagram shown earlier and using an algorithm to determine what range a prediction falls on to on this scale and therefore what the appropriate sentiment label is.




START
prediction_to_sentiment
INPUT prediction as float
Is prediction between 1 and 0.75?
OUTPUT "Very good"
Is prediction between 0.75 and 0.25?
OUTPUT "Good"
Is prediction between 0.25 and -0.25?
Output "Neutral"
Is prediction between -0.25 and -0.75?
Output "Negative"
Output "Very negative"
END subroutine

```
function prediction_to_sentiment(prediction): // Take array of sentiment scores as parameter
    // This block of if statements checks what range the sentiment prediction is in
    if 0.75 < prediction =< 1.00:
        return "Very Positive"

    if 0.25 < prediction =< 0.75:
        return "Positive"

    if -0.25 =< prediction =< 0.25:
        return "Neutral"

    if 0.75 =< prediction < -0.25:
        return "Negative"

    else:
        return "Very Negative"
endfunction

function get_sentiments(predictions): // Take the sentiment predictions of the tweets as a parameter

    sentiments = []
    for i=0 to length(predictions)-1:
        // Use the other function to convert prediction to sentiment
        sentiment = prediction_to_sentiment(predictions[i])

        // Append this sentiment to the list
        sentiment.append(sentiments)
    return sentiments
endfunction
```

The first function called “prediction_to_sentiment” is used for the actual conversion of the prediction score to the sentiment, taking the model’s prediction of a tweet as a parameter. The second function called “get_sentiments” is for handling the list of predictions produced by the model.

I have kept these as two separate functions instead of one so that I can use the “prediction_to_sentiment” function whenever necessary during running of the program for example, I may need to convert the prediction for one tweet instead of a list of tweets which is what “get_sentiments” facilitates.

Now we have three different arrays of corresponding indexes “tweets”, “predictions” and “sentiments”. “predictions” can now be used when the system needs to perform numerical operations and comparisons and “sentiments” can be used for displaying information to the user.

INDEX:	0	1	2	3
“tweets” array:	“I hate this product”	“I like this product	“The product is OK”	“I love this product”
“predictions” array:	-1.0	0.6	0.1	1.0
“sentiments” array:	“Very Bad”	“Good”	“Neutral	“Very Good”

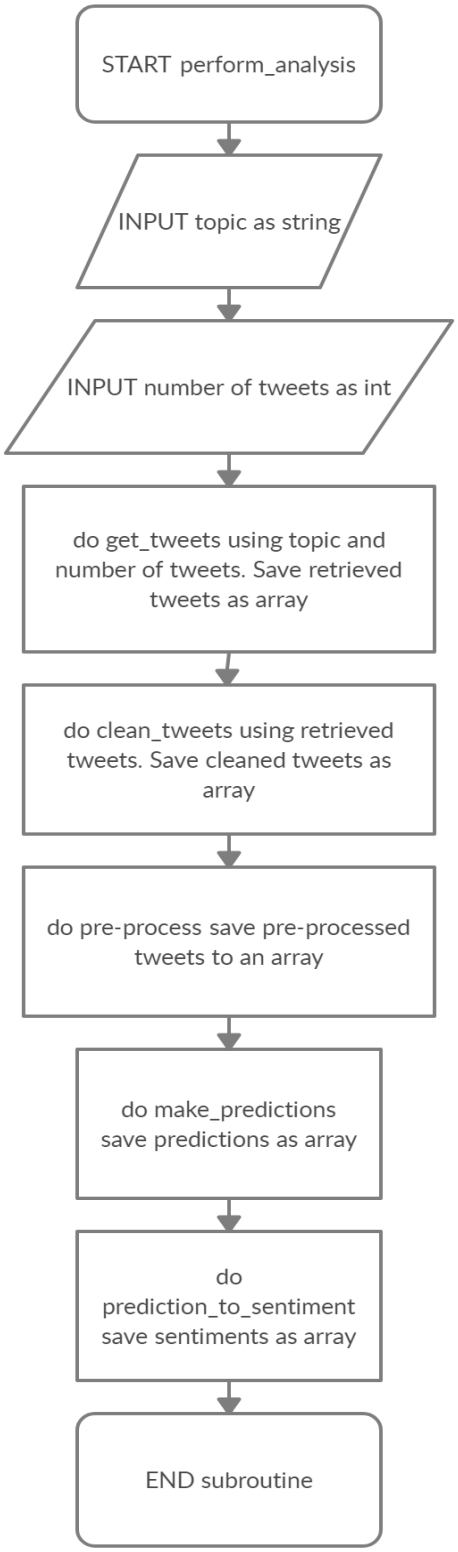
Main analysis function

Now I have created all the necessary functions for providing analysis of a given topic, I need to make a function that connects all of these functions together so that I can use one all-encompassing function to make use of all of those above.

This can be done by providing the topic name (the name of the company or the user topic) and number of required tweets as a parameter and then calling the various functions relating to this data.

This analysis function will then result in the following arrays:

- Tweets array – *this holds all the tweets scraped about the given topic. The number of tweets this array holds is also determined by the user*
- Clean tweets array – *this holds the contents of the tweets array but cleaned so that unnecessary information is removed*
- Pre-processed tweets array – *this holds the tweets that have been cleaned and pre-processed and are now reading to be predicted on by the user*
- Predictions array – *this holds the numerical sentiment predictions for all of the tweets*



- Sentiments array – *this holds the qualitative predictions instead of numerical, e.g holds "positive", "negative" et*

Storing in the database

There are several cases where data will be stored in and retrieved from the database. I will outline some of these below.

Changing password

Below is the algorithm for changing the user's password:

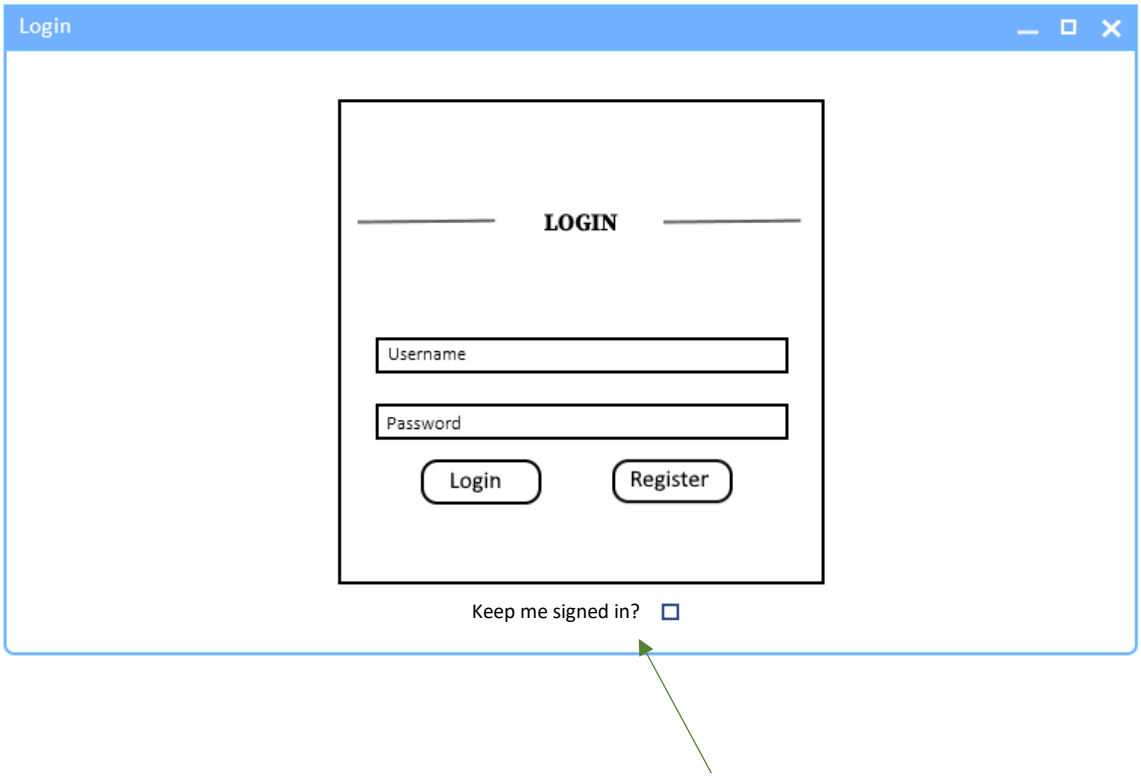
Dashboard

This is the algorithm for retrieving and displaying the data relevant to the company on the dashboard. This is where the user finds out about what people are saying about the company so need

User interface design

Login

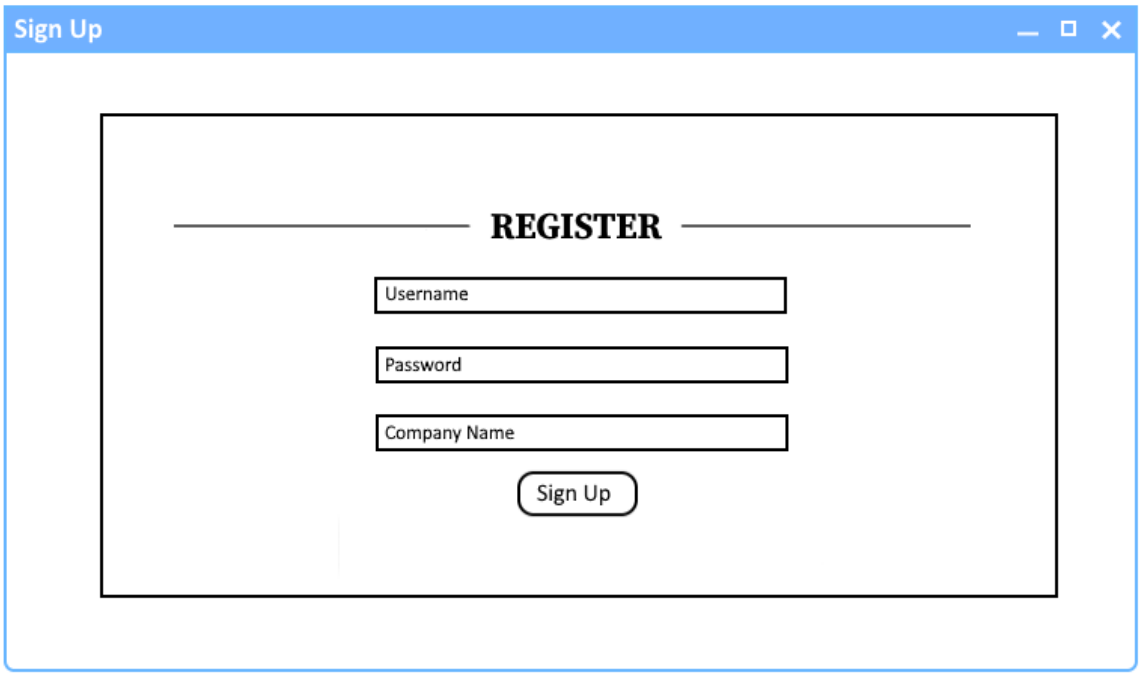
This is the login form for the software. It provides the option to login using the credentials of an existing account and allows the user to create a new account with new details. I have kept the design of the login screen simple as to not create any confusion when first using the software.



The keep me signed in tick box is so that users who are accessing the software frequently have a quick way of accessing the software.

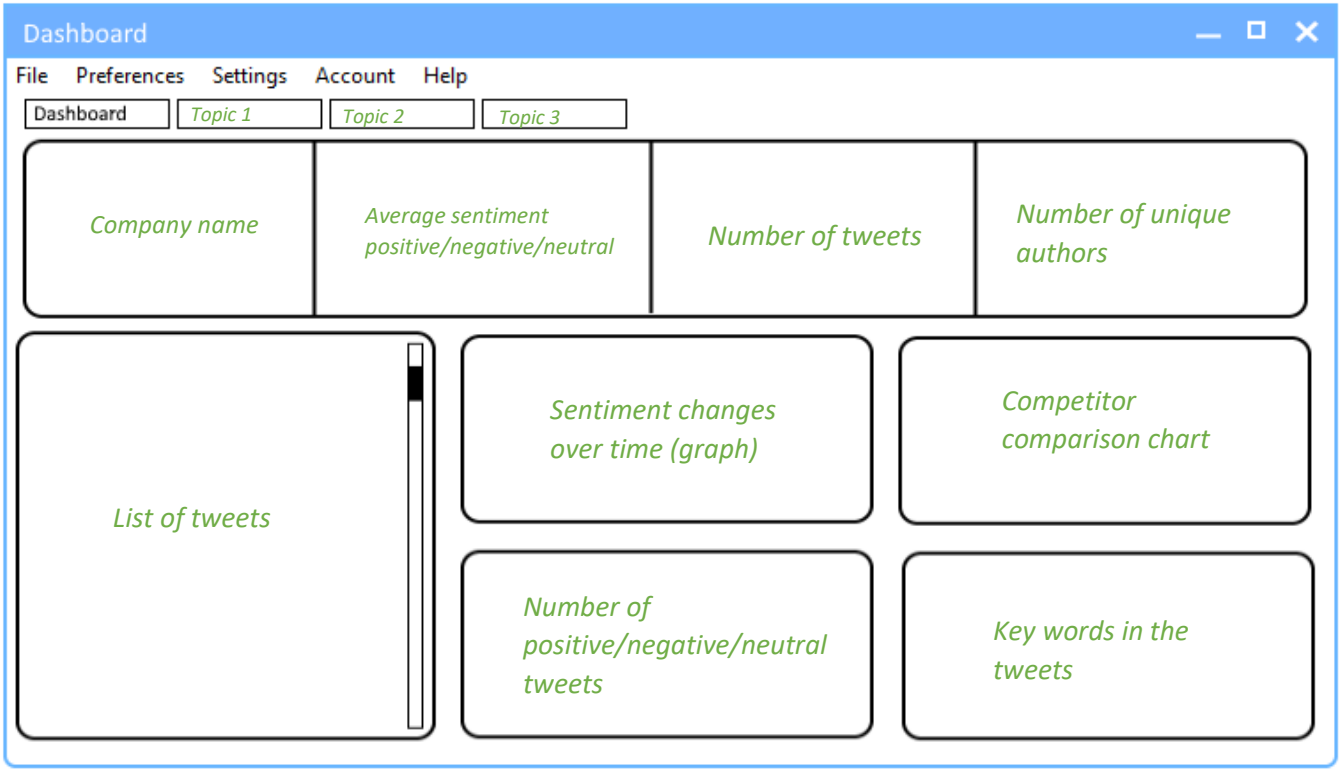
Register

This is the screen displayed for new users who need to make an account. Here I have provided them with the company name as well as standard username and password input fields as means for the system to know what company to provide analysis on.


Sign Up
REGISTER
Username
Password
Company Name
Sign Up

Main Window

This is the main window/ dashboard of the software. This contains each of the main elements that would need to be seen by the user of the software. The green labels company that the user may want to know.



Annotations

Company name – displays the companies name for clarity on what all the data below is about.

Average sentiment – this is the average sentiment of all the tweets made about the company. This is so the user can gauge what most people are saying about the company.

Sentiment changes over time – This is a graph showing how the average sentiment of the company has changed over time. This will be helpful for the user as it would provide a gauge for the reasons behind changes in opinions on the company. An example of this being useful is the user looking at the date a product was released on the graph and look at how that influenced consumer's opinions on the companies.

List of tweets – this is a sample of the tweets used for sentiment analysis for the company. This is included so the user can see what consumers are specifically saying about the company and its products instead of only basing decisions on average opinions. Including this also allows the user to identify issues that consumers are talking about.

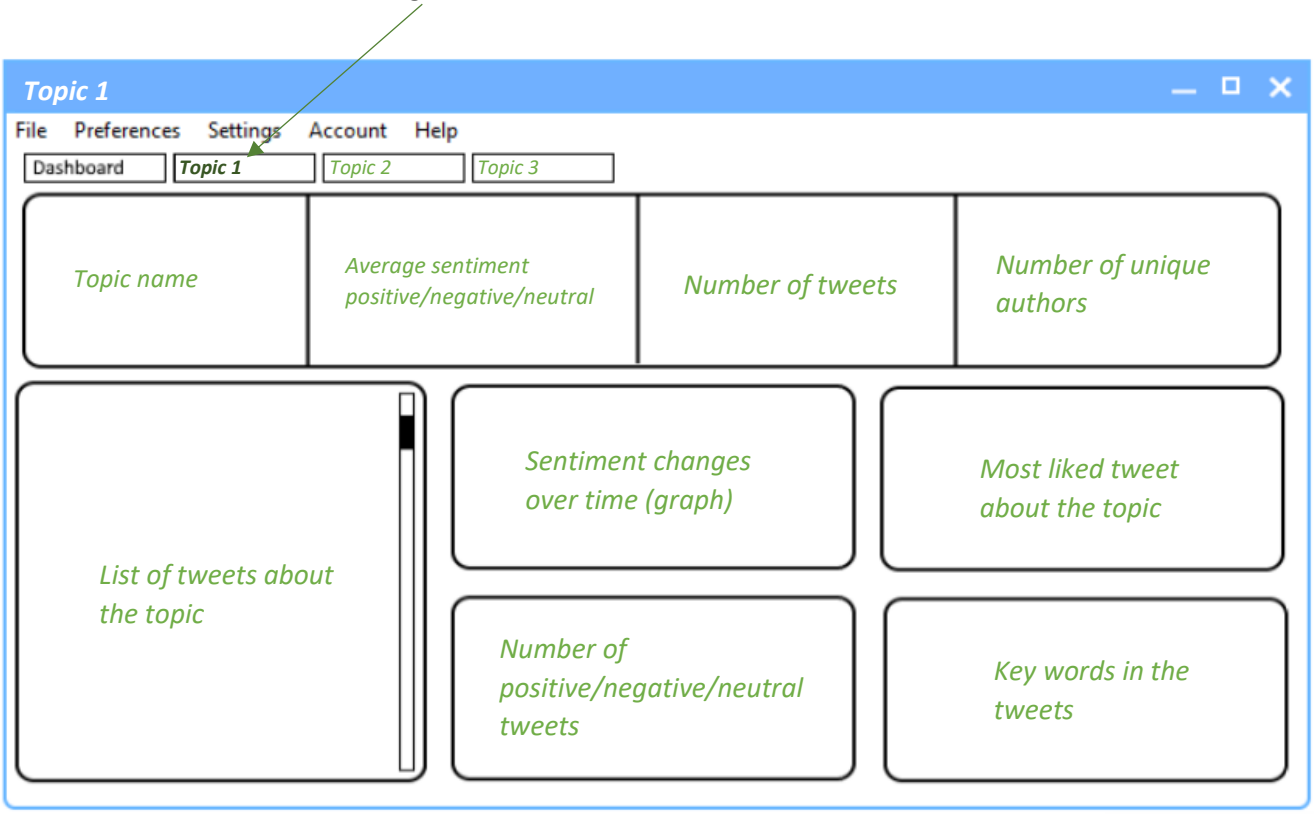
Competitor comparison chart – this is a chart indicating how well the company is doing online in relation to each of its competitors. This chart will be based on the percentage of positive tweets for a user-specified sample size e.g the user will select a sample of 20 and the graph will show what percentage of a sample of 20 tweets about the user's company and its competitors are positive. This will be done on a bar chart.

Key words – this is a list of each of the common words and features of the tweets about the company. This will be helpful for the user to identify any common issues talked about by the consumers. For example, if a lot of the people are talking about the company's customer service, this will be identified in this key words widget.

Topic 1, Topic 2, Topic 3 – these are the tabs where the user can make their own topic sentiment analysis requests on any topic they need. These tabs can be selected at any time during use of the program to view analysis of them. In the image design I have shown 3 tabs labelled "Topic 1", "Topic 2" and "Topic 3". These tabs will have dynamic titles meaning that when the user will make a sentiment analysis request on a topic and a new tab with the topic name as the tab title and the sentiment analysis of the topic as the tab's main content. The user will be able to have as many tabs as required for as many topics they wish to be analysed. (The dashboard tab is the tab displaying the main sentiment analysis on the company.)

Topic Tabs

This is what each topic tab will show when clicked on. Each of the topics requested by the user will have their own window like this with data unique to the topic. The current tab in use is indicated by the title of the of the tab being bold.



As you can see, the topic analysis screen is almost identical to the dashboard screen with similar data widgets so I won't annotate this diagram in its entirety as the widgets generally serve the same purpose as those on the dashboard.

The only difference here is the lack of a competitor comparison chart. This has been replaced with the "most liked tweet about the topic" widget. This widget showing the most liked tweet should give insight to some of the most popular opinions about the topic as this is the tweet the most users have agreed with.

(Again, the titles are dynamic so the where the title of this window is “Topic 1”, when the program is used, this will be replaced with the actual name of the topic. The same goes for the title of the tab which is has been made bold and written in italics to indicate which tab is currently being viewed.)

Drop down menus

File:

- New topic analysis *(this creates a popup window asking for the name of the topic and then a new tab containing the analysis is created)*
- Save dashboard as image *(users may want to send images of the dashboard to co-workers)*
- Print out dashboard *(users may want to keep a hard copy of the dashboard for discussion during meetings)*
- Logout *(Allows the user to logout if several users need to access the software on the same machine)*
- Exit *(allows the user to exit the program)*

Preferences:

- Colour scheme *(allowing the user to customise the colour scheme of the software to suit their preferences will make it more enjoyable to user)*
- Frequency of data logs (from here, the user can tell the system how often it wants it to retrieve more tweets to update the sentiment data. If the user sets this to once a week, the system in background will automatically, retrieve tweet data about the company and the user’s topics and log the analysis results in the database for when the user next uses the software)

Settings:

- Change company (As the dashboard is based on Twitter users’ opinions on the company, the system must know what company the user is affiliated with, this allows them to select their company and change the current company if necessary)

Account:

- Change password (Allows the user to change login password if necessary)

Help:

- How to use the software (Helps the user understand how to operate the software such as how to perform a topic analysis, improves usability)
- About the software (Explains how the software works, giving a brief explanation of the machine learning required for a machine to understand English)

Usability Features

There is a need for good usability features within the software. This piece of software is meant to be used as a tool by business to aid business strategy. For this reason, it is important that the software is easy to understand and use so that the user is making efficient use of the software as to not interfere with any operation of the business.

The user interface designs show that there will be multiple data visualisations and graphs on this screen at one time in the form of charts and graphs. Most of the time spent using the software will be spent viewing these visualisations, so it is essential that they are easily readable. To allow this, I have designed each of these widget boxes to be large enough so that the specific data points within these widget boxes can be clearly read. Also, the window by default will be large on the screen so that everything can be read.

To implement fast access, I have included the “Keep me signed in” tick box with the login screen. This means the user will not need to take the time to sign in every time they want to use the software which may be multiple times in a short period, improving user experience. An issue with this feature is that it reduces the security of the software, but this is not a major concern as again the login feature is more of means to access user unique data rather than provide security and confidentiality.

I have also included a “help” drop-down on the tool bar at the top of the window. This drop-down will include instructions on how to use the software such as how to perform analysis of a necessary topic. This usability feature will make the software more accessible to users and less time will be spent trying to understand how to use the software.

Moreover, the design of the interfaces promotes easy navigation of the software. I have done this by presenting the software on one main window rather than having smaller windows for each element of the software. This reduces clutter and aids the user in finding the necessary information so the user will be spending less time navigating through the software. To maintain readability of the data I have included tabs in the interface designs with the user dashboard and each topic having their own tabs which can be accessed at any time.

In addition, when the user makes a topic sentiment analysis request, it will take the system some time to retrieve, process, make sentiment predictions and display data on the tweets about the topic. It is important that any slowing of the software due to this process is explained to the user to maintain the smooth user experience of the software. This is to avoid causing the user frustration when using the software due to unexplained slowing of the system.

Inputs and outputs

Input	Justification
Topic Request	The user needs to be able to tell the system what they want to find opinions of. For example, if they want to find opinions about their company, they can input the company name.
Login details i.e username and password	There needs to be a login system so that different users can access.
Company name	The user needs to input the name of the company, so the system knows which company to find information about.
Log frequency	The system needs to know how often the analysis data needs to be updated. This may be every day or once a week, for example.
Button inputs: Login button Sign up button	This is so the system can then validate the entered details

Output	Justification
Sentiment score	The model will provide a score which will indicate how positive or negative a piece of text is.
GUI	The program will have a user interface that will include login and signup screens as well as a main window to allow for easier data representation.
Graphs representing analyses data	Graphical representations of data are a much easier way of presenting patterns.

Validation and Maintaining System Integrity

When creating a system like this, there are several instances where input validation is necessary to ensure the system doesn't run into any errors.

Database validation

It is important that there is input validation when handling a database to maintain the integrity of the data it stores. SQL injection could be a potential threat to the system and is a threat during the start of the program where the user either enters login details to be checked in the database or enters registration details to be added to the database. Certain strings when entered can cause the output of data from the database or the deletion of database tables so I need to validate any entered user details to make sure this does not happen.

Another important aspect of input validation for the database is the use of usernames. If multiple users had the same usernames, when the system queries the database, looking for the current user's record, it could accidentally get the other users record as they have the same username. To prevent this, the system could simply check when the user is registering an account that the entered

username has not already been taken. This simple use of validation also helps maintain the integrity of the system.

Another beneficial and easy to implement concept for the system would be encryption within the database. This could be done when a user registers an account, the system could simply encrypt the entered password so if someone somehow managed to retrieve the passwords, they would be unreadable.

Other input validation

During use of the software, the user will mostly be using dedicated user interface elements like buttons to be providing input to the system. These buttons will be provided by the PyQt5 library which I am going to use to create the user interface so these forms of input should already be robust.

There are a few cases where the user does need to provide input such as when choosing how frequently the system to should take logs of the data

Key variables / Classes / Data structures

Variable name	Datatype details	Why is it needed?
Model	NLTK model	This is the model the program will be using to perform sentiment analysis on the provided text and find emotions in the text.
company_name	String. Topic attribute	The system needs to know the name of the company so it can identify the necessary relevant tweets.
username	String. User attribute	The system needs a way to hold the username so that it can query the database.
password	String.	This hold the password the user has entered so it can be compared the password in the database.
stored_password	String	This is the password that is retrieved from the database to be compared with the entered password.

access_token, secret_token	String	These are the keys that provide authentication for access to the Twitter API
num_of_tweets	Int	Used to specify how many tweets the scraper should request from Twitter.
log_freg	Int	This variable is for the system to know how often it should update its tables.

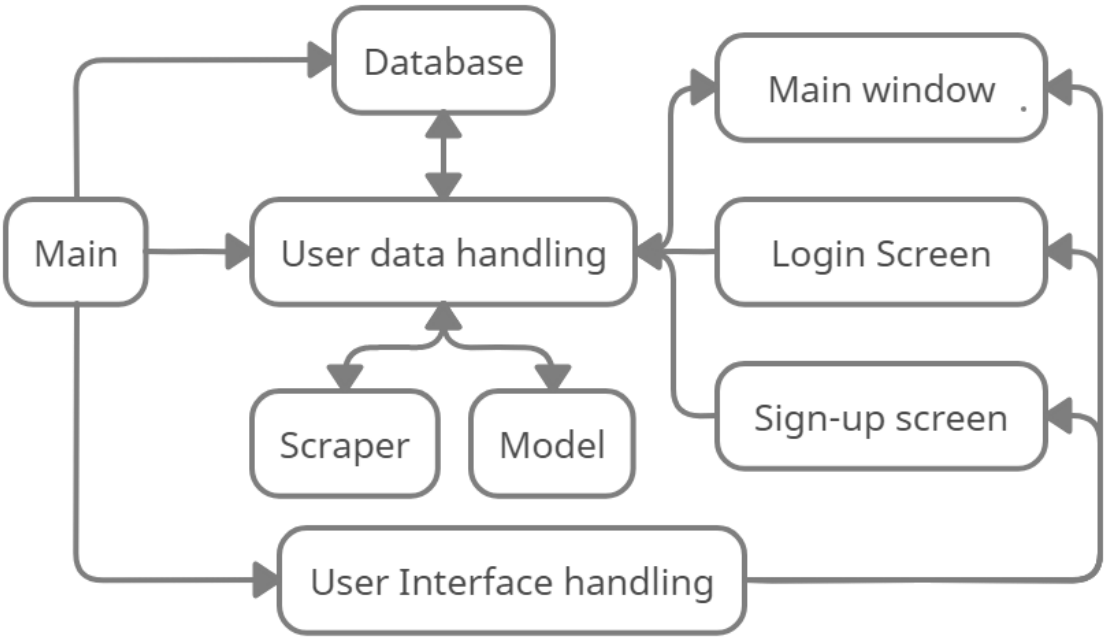
Class name	Datatype details	Why is it needed?
Database	Class	This is class handles the access and use of the database and will be used to insert and retrieve data to the various tables within the database.
User	Class	This class represents a user and all the relevant data about them that may be relevant to use in the duration of the program. This class will need attributes like what the user's name is, what company they belong to and what topics they require analysis of.
Topic	Class	Represents a topic the user enters. Holds data about a specific topic and provides function for interacting with topic data like performing sentiment analysis on the given topic.
Company	Class	Represents the user's company. Holds data about the company as well as methods allowing for the retrieval of tweets about the company and analysis of these tweets.
Scraper	Class	This is the framework for scraping the tweets from twitter. This class will include methods such as a method used for performing the actual scraping of the tweets
Model	Class	This class will be used for handling use of the model. This class will have methods for using the model to make predictions as well as methods that clean and pre-process the text so it can be understood by the model.
Controller	Class	This class is used for handling the graphical user interface required for the application i.e. deciding what is currently shown on the screen.
Login	Class (inherits pyqt5.QDialog)	This is the class that provides the signing up window and all the buttons and text shown on that window. It inherits QDialog so PyQt components like buttons can be embedded on the window.
SignUp	Class (inherits pyqt.QDialog	This is the class that provides the signing up window and all the buttons and text shown on that window.
MainWindow	Class (inherits pyqt5.MainWindow)	The is the primary window that pyqt5 provides to hold each of the widgets and labels that may need to be shown by the application.
Tab	Class (inherits QWidget)	This is the class that acts as a template for every tab that will be on the main window, with each tab holding information on that tab's topic. The use of tabs was suggested in the UI form designs section
DashboardTab	Class (inherits Tab class)	This class represents the dashboard tab on the window and is what determines what will be shown on the user's dashboard when using the software.
TopicTab	TopicTab (inherits Tab class)	This class is the template for topic tabs on the window. This is needed as multiple tabs on different topics will be created so a template for each of these tabs is necessary.

Data structure name	Datatype details	Why is it needed?
tweets	List. Topic attribute	Once tweets are retrieved from Twitter, they need to be held by this variable so that they can be accessed and used by the model
tweet_likes	List. Topic attribute	This array holds the number of likes each corresponding tweet in the “tweets” array has been given. This will be used when displaying the tweets, allowing the user to arrange the displayed tweets in order of most liked.
Tweet_authors	List. Topic attribute	Array holding the authors of the people writing the tweets so they can be shown with the tweets on the user interface.
clean_tweets	List.	This contains the tweets that have been cleaned.
Topic_logs	List. Topic attribute	
pre_processed_tweets	List.	This contains the tweets that have been pre-processed and are now ready for model to make predictions on them.
predictions	List. Topic attribute / Company attribute	This is an array of prediction values the model has made on the tweets. This array holds float values determining how positive or negative its corresponding tweet is.
sentiments	List. Topic attribute / company attribute	This is an array based on the predictions array holding the English sentiments described by the scores represented by the predictions array. i.e holds “Good”, “Bad”, “Neutral” instead of float values. This will be used when displaying results.

System structure overview

Now that I have explained the main components of the solution, I will provide a provisional design of the system structure.

Previously, I used a hierarchal diagram to demonstrate how the different components of the application can be broken down. The following diagram is used to show the relationship between the different components of system.



This diagram shows the flow of data between the different components of the system. For example, the main window will begin running the user interface handler which will in turn display the main window, login screen and sign-up screen.

Justification of system structure and other design choices

System structure:

I have designed the system structure in such a way that minimizes unnecessary interactions and communications between different parts of the program. This is done to ensure that each part of the program is performing the exact task it is intended for and nothing else. It also improves the efficiency of the program as component functions are more specific and aren't performing unnecessary tasks. Minimizing the communication between components will also make code creation easier as problems in one area of the program are unlikely to have any major impacts on the rest of the program.

Modular Design:

The application of this decomposition comes in the form of a modular design. This involves splitting the program into multiple different modules and storing various subroutines in these different modules. These modules can then be accessed to use the subroutines contained within them. This promotes an efficient. code structure which means code will not need to be re-written every time a specific functionality is required. This will be important in the case of this software as several logical elements like the ability to access the database will be required repeatedly during running of the program and access to a module means we can just access this module every time access is required.

Breaking the problem down means that any errors that arise can be addressed individually within each module speeding up development.

This also means that any improvements or added functionality for the program can be implemented effectively as changes that need to be made can occur within the relevant module and not affect the rest of the program. This would be important if I were to implement sentiment analysis of social media platforms other than twitter, for example, so I could simply create a new class intended for this new platform to be analysed.

The alternative for a modular design would be a single file for all of the program's functionality. A file like this could be thousands of lines long, lacking any kind of maintainability. This is because any person accessing the source code will need to look through all of the source code to identify any kind of errors or implement any new features. If there a logic error in the program (an error that causes incorrect outcomes in the program but does not stop the running of the program), such as the program failing to save new login details to the database.

Object Oriented Programming:

Continuing with the concept of efficient code design, Object Oriented Programming promotes modularity by using inheritance. Inheritance allows me to implement new features into the solution without needing to rewrite existing code, further reducing the complexity of the solution.

The need for machine learning

There is an apparent need for machine learning in this project as without any help, the machine has no concept of what words have good or bad connotations.

If we needed to perform sentiment analysis without the use of machine learning, we could use the Lexicon method which involves using a list of words with an associated index quantifying how positive or negative each word was. Then if we were to try and perform sentiment analysis, we could look at which words are used in the sentence and make an average of their sentiment using our index. It would be implemented like this.

```
dictionary words = // (word : score)
[
    "good" : 1,
    "bad" : -1,
    "OK" : 0,
]

sentence1 = "The product is very good"
sentence2 = "The product was good but
customer service was bad"
```

The “words” dictionary holds key words and how positive and negative these words are with a score of “-1” being negative, “1” being positive and “0” being neutral. The program will then count the number of times these key words appear in sentences and find the average score of the words used. This average score indicates the average emotion. For sentence1 in the image, the program would identify “good” as it appears in the dictionary, this would make the average score for the sentence 1

therefore showing the sentence to have “good” sentiment. For sentence 2, “good” and “bad” will be identified and with scores of 1 and -1 respectively, the average will be 0 making it a sentence with an overall neutral sentiment.

This method is simple and easy to implement on a small scale like this but is difficult to scale when considering more vocabulary. This is because for more in-depth analysis, you would need an extremely long list of every single possible words that could appear in text and would need to manually write in all their sentiment scores. This would take a very long time and can be avoided by creating a machine learning model which will learn the sentiment of any words by itself. By using machine learning, most of the time taken to build a script that can predict the sentiment of text will be training the model on a large set of training data.

Approach to testing

To ensure the program is robust I will need to carry out thorough testing of the solution. This means that any bugs or problems in the software that are not yet apparent can be identified and removed before delivering the software to the client in a state which could have a negative impact on the user experience.

Iterative testing

I will be using iterative testing throughout development of the system. This will be done by performing during and at the end of each stage of development. These sections are based on the results of the problem decomposition performed in the top-down design. Each of these parts of the program will be tested individually to ensure that it works without errors and have the necessary integrated features like validation, security. During these iterative tests, I will essentially be

performing unit tests for each module. This table will be used to conduct tests where it is deemed necessary during development.

Test Case	Provided arguments	Expected output	Actual output	Pass/Fail

Summative testing

I will then use summative testing once development of the solution is complete. This summative testing will be used to evaluate the efficacy of the final solution and will need to be very thorough.

Test number	What is being tested?	How is it being tested?	Expected outcome
1	UI formatting	Look for spelling errors across the program. Check UI displayed matches UI designs.	UI matches designs shown earlier in report.
2	Login screen initialisation	Start program.	Login window is displayed correctly upon opening the software.
3	Sign up screen initialisation	Click on “Sign up” button on login window.	Signing up window is displayed.
4	Main window initialisation	Click on “Login” button on login window.	Main window is shown with all necessary visual elements.
5	Data is being loaded correctly	Check all widgets on the main window are displaying the data they should be	Dashboard widgets are showing all necessary data described in design section
6	Tabs showing user topics are displayed as well as dashboard	Check that every topic the user has requested to analyse is present in the tabs shown near the top of the screen	Tabs for each of the user’s topics are present
7	Switching between tabs	Check that you can click on different tabs with each tab showing all the data relevant to that topic without errors when switching back to previous tabs.	All tabs can be clicked on to be accessed with each tab showing it relevant data with no errors.
8	Drop down menu.	Check each element of the drop-down menu at the top of the screen to ensure every option described in the design section is present. Check each available option in the drop-down	Every drop down menu button has its
9			

Development and Testing

The following is a list of the stages of development:

1. Database creation
2. Model creation
3. Scraper creation
4. User data handling
5. User interface
6. Finalizing

Stage 1 – Database creation

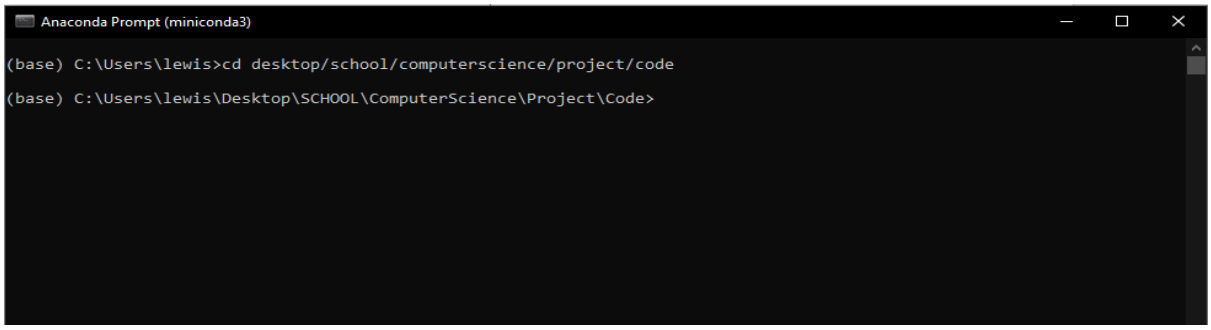
Goals for the stage:

- Create a virtual environment containing a python interpreter which will allow programs created during development to be run using python.
- Create a script that will provide the necessary functionality for working with a database during development. This will include the ability to:
    - Create the database and the necessary tables
    - Register new users and allow for the
    -

Creating the virtual environment using Anaconda

I will be working with Anaconda to create a virtual environment. Anaconda is a specialised tool for data science and includes the ability to create virtual environments which offer access to the necessary dependencies to the local directory.

Below is a screenshot of the Anaconda terminal I can use to create the virtual environment.



Next, I will create the environment using “conda create”. This command as standard includes a python 3.7 interpreter which allows python files to be run, but I also need to include some of the modules that are essential to the project (I will manually install the other modules later in development when I need them.

```
Anaconda Prompt (miniconda3)

(base) C:\Users\lewis\Desktop\SCHOOL\ComputerScience\Project\Code>conda create --prefix ./env pandas numpy matplotlib
Collecting package metadata (current_repodata.json): done
Solving environment: done

## Package Plan ##

  environment location: C:\Users\lewis\Desktop\SCHOOL\ComputerScience\Project\Code\env

  added / updated specs:
    - matplotlib
    - numpy
    - pandas


The following packages will be downloaded:

    package                    |            build
    ---------------------------|-----------------
    ca-certificates-2020.7.22  |                0         125 KB
    certifi-2020.6.20          |           py38_0         157 KB
    freetype-2.10.2            |       hd328e21_0         470 KB
    intel-openmp-2020.2        |              254         1.6 MB
    matplotlib-3.3.1           |                0          25 KB
    matplotlib-base-3.3.1      |   py38hba9282a_0         5.1 MB
    mkl-2020.2                 |              256       109.3 MB
    numpy-1.19.1               |   py38h5510c5b_0          22 KB
    numpy-base-1.19.1          |   py38ha3acd2a_0         3.8 MB
    openssl-1.1.1g             |       he774522_1         4.8 MB
    pandas-1.1.1               |   py38ha925a31_0         7.5 MB
    pip-20.2.2                 |           py38_0         1.8 MB
    pyqt-5.9.2                 |   py38ha925a31_4         3.2 MB
    python-3.8.5               |       he1778fa_0        15.7 MB
    qt-5.9.7                   |   vc14h73c81de_0        72.5 MB
    setuptools-49.6.0          |           py38_0         763 KB
    sqlite-3.33.0              |       h2a8f88b_0         809 KB
    tk-8.6.10                  |       he774522_0         2.7 MB
    wheel-0.35.1               |             py_0          37 KB
    ------------------------------------------------------------
                                           Total:       230.3 MB
```

Now in the folder I am working from I can use this environment to access the necessary modules and use the necessary interpreter to run my python files.

Creating the database

One need for the system is the encryption of user passwords when inserting into the database so I will need to install the cryptography Python library onto my virtual environment as follows:

```
Anaconda Prompt (miniconda3) - conda  install -c anaconda cryptography
(C:\Users\lewis\Desktop\SCHOOL\ComputerScience\Project\Code\env) C:\Users\lewis>conda install -c anaconda cryptography
Collecting package metadata (current_repodata.json): done
Solving environment: done

## Package Plan ##

  environment location: C:\Users\lewis\Desktop\SCHOOL\ComputerScience\Project\Code\env

  added / updated specs:
    - cryptography
```

To begin creating the database, I have created a database class which is using the Sqlite3 library. When the class is imported, the class creates the necessary tables if they do not already exist. This is necessary for new users of the system who will not yet have the database created on their system so trying to access would otherwise cause an error.

```
class Database:
    # Connect to the database
    conn = sqlite3.connect('sentiment.db')
    c = conn.cursor()

    # Add the "users", "user_topics", "topics" table to the database
    c.execute("CREATE TABLE IF NOT EXISTS users(user_ID INTEGER PRIMARY KEY AUTOINCREMENT,"
              "username TEXT,"
              "password TEXT,"
              "company_ID INTEGER,"
              "FOREIGN KEY(company_ID) REFERENCES companies(company_ID))")

    c.execute("CREATE TABLE IF NOT EXISTS companies(company_ID INTEGER PRIMARY KEY AUTOINCREMENT,"
              "name TEXT)")

    c.execute("CREATE TABLE IF NOT EXISTS user_topics(user_ID INTEGER,"
              "topic_ID INTEGER,"
              "FOREIGN KEY(user_ID) REFERENCES users(user_ID)"
              "FOREIGN KEY(topic_ID) REFERENCES topics(topic_ID))")
    c.execute("CREATE TABLE IF NOT EXISTS topics(topic_ID INTEGER PRIMARY KEY AUTOINCREMENT,"
              "name TEXT,"
              "sentiment FLOAT)")

    c.execute("CREATE TABLE IF NOT EXISTS logs(topic_ID INTEGER,"
              "company_ID INTEGER,"
              "sentiment FLOAT,"
              "date DATE,"
              "pos_tweets INT,"
              "num_of_neg_tweets INT,"
              "FOREIGN KEY(topic_ID) REFERENCES topics(topic_ID),"
              "FOREIGN KEY(company_ID) REFERENCES companies(company_ID))")

    # Commit changes to the database
    conn.commit()

    # Disconnect from the database
    c.close()
```

To test if this worked, in my main file, I ran a query to see the current tables in the database.

```
import sqlite3

conn = sqlite3.connect('sentiment.db')
c = conn.cursor()

c.execute("SELECT name FROM sqlite_master WHERE type = 'table'")
print(c.fetchall())
```

Resulting in:

```
[('sqlite_sequence',), ('users',), ('user_companies',), ('companies',), ('user_topics',), ('topics',), ('logs',)]
```

This showed the system can connect to the database as well as create the tables.

When ran a second time, I got the same output. This showed that the system wasn't creating the tables again unnecessarily by recognising that they already exist in the database.

With the foundation of the class is working successfully, I can now begin creating other functionality for the class. I have started with creating the encryption and decryption static methods to be used when saving and retrieving passwords in the database. They both use the cryptography library I installed earlier to allow for encryption and access an encryption key stored in an external file to do this.

```
@staticmethod
def encrypt(password):

    # Import the externally saved encryption key
    from keys import encryption_key

    # Return the encrypted password
    f = Fernet(encryption_key)
    password_in_bytes = password.encode()
    encrypted_password = f.encrypt(password_in_bytes)
    return encrypted_password

@staticmethod
def decrypt(encrypted_password):

    # Import the externally saved encryption key
    from keys import encryption_key
    f = Fernet(encryption_key)

    # Return the decrypted password
    decrypted_password = f.decrypt(encrypted_password).decode()
    return decrypted_password
```

When trying to import cryptography so that these functions would work, I ran into this error which would not allow me to use the cryptography library:

```
    from cryptography.hazmat.bindings._openssl import ffi, lib
ImportError: DLL load failed: The specified procedure could not be found.
```

I found a solution online at https://github.com/pyca/cryptography/issues/4011. To fix the problem I simply needed to update each of the packages within the environment as there must have been outdated dependencies.

```
Anaconda Prompt (miniconda3) - conda  update --all
(C:\Users\lewis\Desktop\SCHOOL\ComputerScience\Project\Code\env) C:\Users\lewis>conda update --all
```

Now when the script is run, although there is not yet an output, the error is no longer produced.

To test that the encrypt and decrypt methods are working I ran the code below in the main file of the program.

```
password = "password"

encrypted_password = Database.encrypt(password)
print(f"encrypted password: {encrypted_password}")

decrypted_password = Database.decrypt(encrypted_password)
print(f"decrypted password: {decrypted_password}")
```

This resulted in this output:

```
encrypted password: b'gAAAAABgBq36Sa_4Sf7zX5h8FCp6uUheJSwmAV4oig7HAFt4C5xHBdgPExU9QCJM3Xi6ie5p8vUCLQr_tzWetFVmhiblAjCyxw=='
decrypted password: password
```

This shows that both the encrypt and decrypt functions are working correctly.

Below is the method to add a new user to the database. When adding the new user, the system needs to know their username, password, and name of the user's company so the system knows which company there will need to be analysis to be performed on. I will explain the ongoings of the method below.

```python
@classmethod
def new_user(cls, username, password, company):

    # Check if they have entered a valid username
    if len(username) < 2 or len(username) > 30:
        return "Please enter a valid username"

    # Check if they have entered a valid company name
    if len(company) < 2 or len(company) > 30:
        return "Please enter a valid company name"

    # Check if the password is long enough
    valid_length = True
    if len(password) < 4:
        valid_length = False

    # Check if there is at least one special character in the password
    special_chars = "!\"£$%^&*(){}[];\':@<>,./?\|"
    contains_special = False
    for char in special_chars:
        if char in password:
            contains_special = True
            break

    # Check if there is at least one uppercase character
    contains_upper = False
    for char in password:
        if char.isupper():
            contains_upper = True
            break

    # Don't add the new details if the password isn't strong enough
    if not (valid_length and contains_special and contains_upper):
        return "Password not strong enough"

    # Query user table to see if the username exists
    conn = sqlite3.connect('sentiment.db')
    c = conn.cursor()
    c.execute("SELECT * FROM users WHERE username = ?", (username,))
    username_exists = c.fetchone()

    # If the username exists, don't add the new user details
    if username_exists:
        c.close()
        return "Username already exists"

    # Encrypt the password
    encrypted_password = cls.encrypt(password)

    # Check if the company already exists in the companies table
    c.execute("SELECT company_ID FROM companies WHERE name = ?", (company,))
    company_exists = c.fetchone()

    if company_exists:
        # Get the ID of the company the user has entered
        companyID = company_exists[0]

        # Add the new user record
        c.execute("INSERT INTO users (user_ID, username, password, company_ID) VALUES (NULL, ?, ?, ?)",
                  (username, encrypted_password, companyID))

    else:
        # Add the new company record
        c.execute("INSERT INTO companies (company_ID, name) VALUES (NULL, ?)", (company,))

        # Get the ID of this new company
        c.execute("SELECT company_ID FROM companies WHERE name = ?", (company,))
        companyID = c.fetchone()[0]

        # Add the new user record
        c.execute("INSERT INTO users (user_ID, username, password, company_ID) VALUES (NULL, ?, ?, ?)",
                  (username, encrypted_password, companyID))

    conn.commit()
    c.close()

    return
```

Firstly, the username and company name are validated by checking the length of these strings. The password is then validated by checking if the password is strong enough to be used in the database. This system classifies a strong password as being one that is at least 8 characters long, contains at least one uppercase character and contains at least one special character.

The next piece of validation is to check whether the tables already contain a user with this username. With this validation in place, it adds an extra amount of robustness when querying the tables.

Once the password has been validated, the password is encrypted using the function described prior.

It then checks if the company the user has entered is already in the companies table. If not, it adds this company as a new record to the companies table and adds the new user record with the corresponding id of the new company record. If this company does already exist, it gets the id of the existing company and then includes this in the company_id field when the adding the new user record.

To test this, I provided user details using this syntax in the main function.

```
def main():
    database = Database()
    database.connect()
    print(inst.new_user("username", "password", "company"))
    database.disconnect()
```

These are the results.

Test case	Provided arguments	Expected output	Actual Output	Pass/Fail
Valid username	“username”, “Password1!”, “company”	None	`None`	Pass
Invalid username	“u”, “Password1!”, “company”	“Please enter a valid username”	`Please enter a valid username`	Pass
Invalid company name	“username”, “Password1!”, “c”	“Please enter a valid company name”	`Please enter a valid company name`	Pass
Password too weak	“username”, “password”, “company”	“Password not strong enough”	`Password not strong enough`	Pass
Username already exists	“username”, “Password1!”, “company”	“Username already exists”	`Username already exists`	Pass

As all test cases have passed, this shows the new user function to be successfully working at a surface level. To see if the data was correctly inserted in the database, I wrote the following lines in the main file which will retrieve the password from the record which was correctly inserted when performing the testing above.

```
conn = sqlite3.connect('sentiment.db')
c = conn.cursor()

c.execute("SELECT password FROM users WHERE username = ?", ("username",))
encrypted_password = c.fetchone()[0]
decrypted_password = Database.decrypt(encrypted_password)
print(f"encrypted password: {encrypted_password}")
print(f"decrypted passsword: {decrypted_password}")
```

This is the output, displaying the stored encrypted password and the password once decrypted:

```
encrypted password: b'gAAAAABgBHhl3yV5wJfBXpFm37rr_FRK2HH7IyjGei--0VlnKv-TkHVuaWqgU6EJmjvVzaqiKMVhu_sE6ekBNCN-8yrjDrVKNQ=='
decrypted passsword: Password1!
```

This simple test shows that the user details were correctly inserted into the database and can be retrieved from the database. It also shows that the details could also be successfully encrypted when inserting and decrypted when retrieving.

Now I will test if the companies are being stored correctly. The following code adds a new user to the database and then gets the company this user record is associated with. This is testing whether the system is able to recognise that the entered company already exists and references the existing company in the user record instead of creating a new company record.

```
import sqlite3

conn = sqlite3.connect('sentiment.db')
c = conn.cursor()

Database.new_user("username", "Password1!", "company1")
c.execute("SELECT company_ID FROM users WHERE username = ?", ("username",))
company_ID = c.fetchone()[0]
print(f"Company ID of the entered company is {company_ID}")
c.execute("SELECT name FROM companies WHERE company_ID = ?", (company_ID,))
print(f"Name of this company is {c.fetchone()[0]}")

c.close()
```

Test case	Provided arguments	Expected output	Actual Output	Pass/Fail
New user entering new company	“username”, “Password1!”, “company1”	“ Entered company ID: 1”	`Company ID of the entered company is 1` `Name of this company is company1`	Pass

		“Name of this company: company1“		
New user entering same company name.	“username2”, “Password1!”, “company1”	“Entered company ID: 1” “Name of this company: company1“	Company ID of the entered company is 1 Name of this company is company1	Pass
A new user entering a different new company	“username3”, “Password1!”, “company2”	“New ID: 2” “Name of this company: company2“	Company ID of the entered company is 2 Name of this company is company2	Pass

This shows the system can handle multiple users from the same company, reducing data redundancy by each of these users at the same company accessing the same company record instead of each of them having their own company record with the same company data.

Now, I need to make the function which authenticates users trying to login to their account. This will need to be able check the entered password against the password stored in the database.

```
def authenticate(self, username, entered_password):

    # Connect to database
    self.connect()

    # Get the stored password
    self.c.execute("SELECT password FROM users WHERE username = ?", (username,))
    stored_password = self.c.fetchone()[0]
    self.disconnect()

    # Decrpyt the stored password
    decrypted_password = self.decrypt(stored_password)

    # Check if the entered and stored passwords match
    if decrypted_password == entered_password:
        return "Details valid"
    else:
        return "Invalid username or password"
```

I tested this function using similar means as testing the “new_user” function.

```
import sqlite3
from database import Database

conn = sqlite3.connect('sentiment.db')
c = conn.cursor()

print(Database.authenticate("username", "password!"))
```

Test case	Provided arguments	Expected output	Actual Output	Pass/Fail
Valid login credentials	“username”, “password1!”	“Details valid”	`Details valid`	Pass
Invalid login credentials (password is wrong)	“username”, “wrongpassword”	“Invalid username or password”	`Invalid username or password`	Pass
Invalid login credentials (username does not exist)	“username1”, “password1!”	“Invalid username or password”	`stored_password = self.c.fetchone()[0]` `TypeError: 'NoneType' object is not subscriptable`	Fail

This error suggested that the function is unable to handle cases where the username does not exist in the users table. To fix this, I implemented username validation which would occur before password validation. I also simplified the outputs to just “True” or “False” which should make handling the results of the authentication easier. I made the function a class method to eliminate the need for instantiation which makes it easier to use in other classes.

```
@classmethod
def authenticate(cls, username, entered_password):

    # Connect to database
    conn = sqlite3.connect('sentiment.db')
    c = conn.cursor()

    # Check if record with this username exists
    c.execute("SELECT count(username) FROM users WHERE username=?", (username,))
    if c.fetchone()[0] == 0:
        c.close()
        return False

    # Get the stored password
    c.execute("SELECT password FROM users WHERE username = ?", (username,))
    stored_password = c.fetchone()[0]

    c.close()

    # Decrpyt the stored password
    decrypted_password = cls.decrypt(stored_password)

    # Check if the entered and stored passwords match
    if decrypted_password == entered_password:
        return True
    else:
        return False
```

I tested the function in the exact same way to see if the changes fixed the issue.

Test case	Provided arguments	Expected output	Actual Output	Pass/Fail
Valid login credentials	“username”, “password1!”	True	True	Pass
Invalid login credentials (password is wrong)	“username”, “wrongpassword”	False	True	Pass
Invalid login credentials (username does not exist)	“username1”, “password1!”	False	True	Pass

Looking at the success criteria, I also need to add the ability for the user to change their password. I have created the following class method to do this:

```
@classmethod
def change_password(cls, username, new_password):

    # Connect to the database
    conn = sqlite3.connect('sentiment.db')
    c = conn.cursor()

    # Encrypt the new password
    encrypted_password = cls.encrypt(new_password)

    # Update the record with the new password
    c.execute("UPDATE users SET password = ? WHERE username = ?", (encrypted_password, username))
    conn.commit()
    c.close()
```

To test this, I wrote this small temporary script within the main file:

```
import sqlite3
from database import Database


def get_password(username):
    conn = sqlite3.connect('sentiment.db')
    c = conn.cursor()
    c.execute("SELECT password FROM users WHERE username = ?", (username,))
    stored_password = c.fetchone()[0]
    c.close()

    return Database.decrypt(stored_password)


username = "Admin"
print(f"Password before: {get_password(username)}")

Database.change_password(username, "new_password")
print(f"Password after: {get_password(username)}")
```

This code gets the password from the user record with the username “Admin” and will output the password. It then uses the “change_password” method I just created to change the password of this user record. It then gets the password from the “Admin” record and outputs what the current password is now. When first run, this was the result:

```
Password before: Admin
Password after: new_password
```

Showing the change to be successful, and therefore the function to work as intended.

It also might be helpful if the user is able to change which company the are known to be a part of. I can implement this using similar techniques used when changing the password:

```python
@classmethod
def change_company(cls, username, new_company):
    # Connect to the database
    conn = sqlite3.connect('sentiment.db')
    c = conn.cursor()

    # Check if the company already exists in the companies table
    c.execute("SELECT company_ID FROM companies WHERE name = ?", (new_company,))
    new_company_exists = c.fetchone()

    if new_company_exists:
        # Get the ID of the company the user has entered
        newCompanyID = new_company_exists[0]

        # Update the record with the new company
        c.execute("UPDATE users SET company_id = ? WHERE username = ?", (newCompanyID, username))

    else:
        # Add the new company record
        c.execute("INSERT INTO companies (company_ID, name) VALUES (NULL, ?)", (new_company,))

        # Get the ID of this new company
        c.execute("SELECT company_ID FROM companies WHERE name = ?", (new_company,))
        newCompanyID = c.fetchone()[0]

        # Update the record with the new company
        c.execute("UPDATE users SET company_id = ? WHERE username = ?", (newCompanyID, username))

    conn.commit()
    c.close()
```

Not only is this able to change the associated company in a user's record, it will use a similar method as before to see if the company already exists and use the ID of the existing company.

An important method I haven't added yet is the ability to make a new topic. It may also be useful for the user to be able to remove topics, so I will add that now as well.

```
@staticmethod
def new_topic(userID, topicName, topicSentiment=None):

    # Connect to the database
    conn = sqlite3.connect('sentiment.db')
    c = conn.cursor()

    # Check if the topic already exists in the topics table
    c.execute("SELECT topic_ID FROM topics WHERE name = ?", (topicName,))
    topic_exists = c.fetchone()

    if topic_exists:
        # Get the ID of the existing topic the user has entered
        topicID = topic_exists[0]

        # Link this topic with the user's record
        c.execute("INSERT INTO user_topics (user_ID, topic_ID) VALUES (?, ?)", (userID, topicID))

    else:
        # Add the topic to the topic table
        c.execute("INSERT INTO topics (topic_ID, name, sentiment) VALUES (NULL, ?, NULL)", (topicName,))

        # Get the ID of this new topic
        c.execute("SELECT topic_ID FROM topics WHERE name = ?", (topicName,))
        topicID = c.fetchone()[0]

        # Link this topic with the user's record
        c.execute("INSERT INTO user_topics (user_ID, topic_ID) VALUES (?, ?)", (userID, topicID))

    conn.commit()
    c.close()

    return topicID

@staticmethod
def remove_topic(userID, topicID):
    # Connect to database
    conn = sqlite3.connect('sentiment.db')
    c = conn.cursor()

    # Remove the record linking the user account to the topic and save changes
    c.execute("DELETE FROM user_topics WHERE user_ID = ? AND topic_ID = ?", (userID, topicID))
    conn.commit()
    c.close()
```

Now I will add the functions required for making data logs in the database. This will be necessary for storing data over time. Below is the function for making logs:

```
@staticmethod
def make_log(topic):
    # Connect to the database
    conn = sqlite3.connect('sentiment.db')
    c = conn.cursor()

    # Get the current date
    date = datetime.now()
    date = date.strftime("%d/%m/%Y")

    # Check whether it is a company or topic log being made
    if topic.type == "topic":
        # Insert topic log data
        c.execute(
            "INSERT INTO logs (topic_ID, company_ID, sentiment, date, pos_tweets, neg_tweets) VALUES (?, NULL, ?, "
            "?, ?, ?)", (topic.id, topic.currentSentiment, date, topic.posTweets, topic.negTweets))

    elif topic.type == "company":
        # Insert company log data
        c.execute(
            "INSERT INTO logs (topic_ID, company_ID, sentiment, date, pos_tweets, neg_tweets) VALUES (NULL, ?, ?, "
            "?, ?, ?)", (topic.id, topic.currentSentiment, date, topic.posTweets, topic.negTweets))

    conn.commit()
    c.close()
```

It takes the topic/company that is to be logged in the database and checks whether it is a company or a user topic. It then stores the id of the topic/company along with the date the log is made as well as the sentiment at the time of making the log as well as the number of positive and negative tweets found about the company/topic.

Now I can add the function which retrieves the logs of a given topic/company which will be necessary to show changes in information over time. This is the finished function:

```
@staticmethod
def get_logs(topic):
    # Connect to the database
    conn = sqlite3.connect('sentiment.db')
    c = conn.cursor()

    # Check whether it is a company or topic log being retrieved
    if topic.type == "topic":
        # Get the logs of the topic with the provided id
        c.execute("SELECT * FROM logs WHERE topic_ID = ?", (topic.id,))

    elif topic.type == "company":
        # Get the logs of the company with the provided id
        c.execute("SELECT * FROM logs WHERE company_ID = ?", (topic.id,))

    # Get the results of the query
    logs = c.fetchall()

    # Remove company/topic id
    logs = [log[2:] for log in logs]

    c.close()

    return logs
```

Like the function to make the logs, it takes the topic/company as a parameter, checking whether it is a company or topic and then retrieves all of the logs with the matching id of this topic/company. The returned result will be a list containing the date, sentiment and number of tweets of the topic/company at the time that the log was made.

The “get_user_info” method takes the user’s username as a parameter and then finds the company they belong to and their ID which is foreign key and will be used to find out the user’s data from the other tables like the log table or the companies and topics tables.

```
@staticmethod
def get_user_info(username):

    # Connect to the database
    conn = sqlite3.connect('sentiment.db')
    c = conn.cursor()

    # Get the user record
    c.execute("SELECT user_ID, company_ID FROM users WHERE username = ?", (username,))
    data = c.fetchone()
    userID = data[0]
    companyID = data[1]

    c.execute("SELECT name FROM companies WHERE company_ID = ?", (companyID,))
    company = c.fetchone()[0]

    c.close()

    # Return the user id and user company
    return userID, company, companyID
```

I have also made the “get_user_topics” method which will use the user’s ID found by the “get_user_info” method and finds all of the user’s topics that have been stored in the database and are linked to their profile.

```
@staticmethod
def get_user_topics(user_id):
    # Connect to the database
    conn = sqlite3.connect('sentiment.db')
    c = conn.cursor()

    # Get all of the user's associated topics
    c.execute("SELECT topic_ID FROM user_topics WHERE user_ID = ?", (user_id,))
    topic_ids = [i[0] for i in c.fetchall()]

    # Get all of the topics using the retrieved IDs
    topics = []
    for index in topic_ids:
        c.execute("SELECT topic_ID,name FROM topics WHERE topic_ID = ?", (index,))
        current_topic = [topic for topic in c.fetchone()]
        topics.append(current_topic)

    c.close()

    return topics
```

Stage 1 – Reflection

What has been completed?

In the stage, I created the necessary features within the application to allow for interaction with a database. The main way I did this was using a main database class which includes the ability to: create the database (if it does not currently exist on the user’s computer); functions for encryption and decryption of user passwords; adding new users to the database who are registering a new account; adding a new user topic to the database and associating that with a specific user record and some other features. I also added some other data handling functionality such as classes which involved user and user topics/company data manipulation.

Overview of the current prototype

Currently, the software is comprised of one Python script able to access and make use of a database. I also have a main file containing a main function which will be used as the main way to control the system. When the main file is run, it imports the database Python file and uses it to check whether the database exists yet on the user’s machine, it then creates the database if not. On its own, this is all the system does, but the methods provided by the database class will be used in future areas of the system.

What has been tested and how?

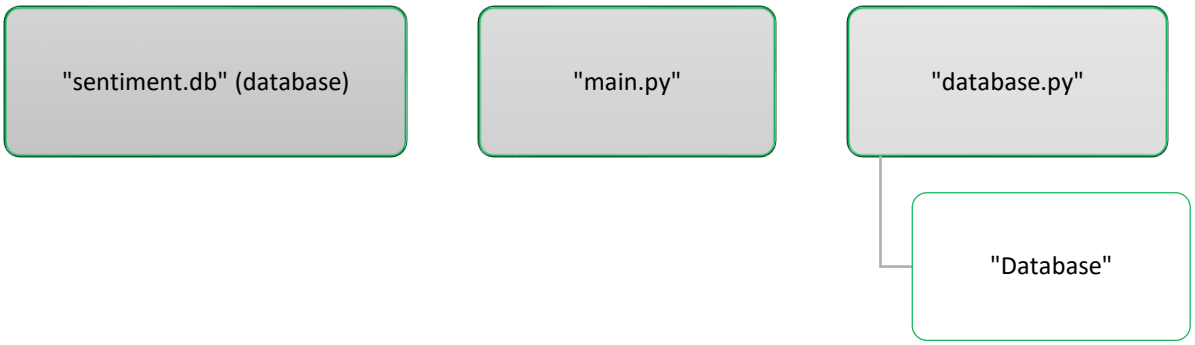
During development, I performed tests on the major functions of the database class to ensure that they working as intended. One test involved testing the functions that provide text encryption and decryption. Testing these functions simply involved passing a simple phrase into the encrypt function which then provided an unreadable string of characters which was then passed into the decrypt function which then returned the original entered phrase.

Links to the success criteria

- *"User authentication in the form of user login and password. This will be validated in a database."*
- *"Users will be able to create a new account, entering login details and the company they belong to."*
- *"The user will be able to select the option to change the password to their account."*

Each of these aspects of the program have now been completed. Now, they each require integration within a user interface

Current system structure (new items have a green outline)



Stage 2 – Creating the Machine Learning model

Goals for the stage:

- Find an appropriate dataset to train the model from
- Train and test the model using the test set of data and evaluate its efficacy
- Make any changes that could improve performance of the model
- Save the model to an external file
- Create a script (to be included within the application) that must:
    - Load the saved model
    - Make predictions on provided text

Finding an appropriate dataset to train the model

Kaggle is a website created to provide a wide range of datasets for various data science applications. It is important that the dataset I choose from Kaggle contains appropriate data for the purpose. The size of the dataset is also important as training a model on a dataset that is too small will mean it may miss certain patterns in text that could improve its accuracy.

The dataset I plan to use is the Sentiment140 dataset which contains 1.6 million real tweets that have been scraped from online. The size of this dataset should be sufficient to create an adequately accurate model.

I am using Google Colab to create the model. Colab is an online platform created by Google to provided notebooks for data science. I am using Colab as Google provides their own Graphical Processing Units (GPUs) for use when training models due to massive parallel processing needed to do this. This notebook I am creating the model in will not be included in the application files as I am only using it to create a model which I can then save externally and use where needed in the application. Colab also provides multiple pre-installed libraries that I will need to install locally to my machine if I need to use them in the program files. (For the development of the actual code of the program I have, and will continue to use, PyCharm)

Model creation

As the code used to create the model in Google Colab will not be used in the final application, I have decided against its inclusion in this report. What I essentially had to do to create the classifying model was use the Sentiment140 dataset to train and test a Bayesian classifier, which upon creation, I then downloaded from the cloud and is now ready to be imported and used in the Python application.

The libraries I used to create the model were all provided by Google Colab and aren't yet installed locally to my computer. Some of which I will need in the application files to access and make use of the model. First, I need to install the libraries I will need for cleaning and preprocessing including "scikit-learn" and "nltk".

Now this is done, I can start creating the "Model" in a new python file called "model.py". The first thing the class will need to do is load the model for use during the running of the application. This can be done by using the "pickle" library which comes pre-installed with Python.

In the file "model.py":

```
class Model:
    # Create the tools used for cleaning
    stop_words = set(stopwords.words('english'))

    slang = {
        'u': 'you',
        'r': 'are',
        'some1': 'someone',
        'yrs': 'years',
        'hrs': 'hours',
        'mins': 'minutes',
        'secs': 'seconds',
        'pls': 'please',
        'plz': 'please',
        '2morow': 'tomorrow',
        '2day': 'today',
        '4got': 'forget',
        '4gotten': 'forget',
    }

    # Create a list of emojis to remove
    emojis = re.compile(pattern="["u"\U0001F600-\U0001F64F"
                                u"\U0001F300-\U0001F5FF"
                                u"\U0001F680-\U0001F6FF"
                                u"\U0001F1E0-\U0001F1FF"   "]+",
                        flags=re.UNICODE)

    # Create the tools used for pre-processing
    lemmatizer = WordNetLemmatizer()

    tokenizer = TweetTokenizer(reduce_len=True)

    # Load the model to be used
    model_file = open("sentiment_model.pickle", 'rb')
    model = pickle.load(model_file)
    model_file.close()
```

I have also added some of the necessary tools for cleaning and pre-processing that will be used to prepare the tweets to be analysed by the model.

Now I will add the class' methods. This class needs to be able to clean, pre-process and make predictions on tweets so each of these features will have their own methods within the "Model" class as follows.

```python
@classmethod
def clean(cls, text):
    # Remove punctuation
    text = text.translate(str.maketrans('', '', string.punctuation))

    # Remove numbers
    text = text.translate(str.maketrans('', '', '0123456789'))

    # Remove stop words
    text = [item.lower() for item in text.split() if item not in cls.stop_words]

    # Remove slang/abbreviations
    text = [cls.slang[item] if item in cls.slang.keys() else item for item in text]

    # Remove links
    text = [re.match('(.*?)http.*?\s?(.*?)', str) for item in text]

    # Join the text back together
    cleaned_text = ' '.join(text)

    return text

@classmethod
def pre_process(cls, text):
    # Lemmatize words
    text = [cls.lemmatizer.lemmatize(item) for item in text]

    # Tokenize words
    text = cls.tokenizer.tokenize(' '.join(text))

    # Make the data into the structure readable by the model
    text = dict([item, True] for item in text)

    return text

@classmethod
def make_model_prediction(cls, prepared_text):
    # Use the model to make a prediction
    prediction = cls.model.classify(prepared_text)

    return prediction

@staticmethod
def get_sentiment(predictions):
    # Convert predictions to readable sentiments
    sentiments = ["Positive" if prediction == 1 else "Negative" for prediction in predictions]

    return sentiments
```

The “clean” method, as indicated by the comments, removes punctuation, numbers, stop words (words like “it”, “or”, “they”, “the” etc which don’t convey opinion), replaces slang with their actual meaning and then removes links to other sites which also obviously don’t convey opinion.

The “pre_process” method then follows what was explained in the design section; lemmatizing of text which involves changing similar words to the same word for the sake of consistency; tokenization of text which involves creating tokens out of the remaining words in the text after

cleaning and pre-processing. It the converts these tokens into a dictionary which is the only readable for of input for the particular type of model I am using.

I have also added a function to convert the values the model has predicted to a readable sentiment score as described in the design section.

To test the current state of the class, I wrote these lines at the bottom of the file. These lines should use the class I have created to clean and pre-process the tweets and then create a list comprised of the predictions on the given tweets.

```
tweets = ["I love ice cream", "I hate ice cream"]

# Create a list of clean tweets
cleaned_tweets = [Model.clean(tweet) for tweet in tweets]

# Create a list of pre-processed-tweets
pre_processed_tweets = [Model.pre_process(tweet) for tweet in cleaned_tweets]

# Create a liost of predictions for the tweets
predictions = [Model.make_model_prediction(tweet) for tweet in pre_processed_tweets]

sentiments = Model.get_sentiment(predictions)

print(predictions)
print(sentiments)
```

This is the result:

Test case	Provided arguments	Expected output	Actual Output	Pass/Fail
A positive phrase followed by a negative phrase.	[“I love ice cream”, “I hate ice cream”]	[1,0] [positive, negative]	`[1, 0]` `['Positive', 'Negative']`	Pass

As this was successful, I then implemented this sequence of code used to test into a method within the model class so that every time I need to make a prediction on text I don’t need to write out all of these lines. This function called “make_tweet_predictions” looked like this within the model class.

```
@classmethod
def make_tweet_predictions(cls, tweets):
    # Create a list of clean tweets
    cleaned_tweets = [cls.clean(tweet) for tweet in tweets]

    # Create a list of pre-processed tweets
    pre_processed_tweets = [cls.pre_process(tweet) for tweet in cleaned_tweets]

    # Create a list of predictions for the tweets
    predictions = [cls.make_model_prediction(tweet) for tweet in pre_processed_tweets]

    # Get the list of sentiments
    sentiments = get_sentiments(predictions)

    return predictions, sentiments
```

This means I now only need to give this single method a series of tweets to get predictions.

To test if this change worked, I replaced my previous lines I used to test the functions with this, giving the method the same list of tweets as an argument:

```
tweets = ["I love ice cream", "I hate ice cream"]

# Get predictions and sentiments of the text
predictions, sentiments = Model.make_tweet_predictions(tweets)

print(predictions)
print(sentiments)
```

This also resulted in an output of:

```
[1, 0]
['Positive', 'Negative']
```

This proved the change to be successful.

This means the model component of the application is now complete and is now ready to be integrated.

Stage 2 – Reflection

What has been completed?

In this stage, I created a machine learning model able to classify the sentiment of text with a considerable level of accuracy. Next, I created a Python script to make use of this model. The script contains a class called “Model” which provides the necessary functionality to make use of this model. It can clean and pre-process text data and then load and use the model to make predictions on this text, providing a score describing how positive or negative it predicts the text is.

Overview of the prototype as a whole

At this point in development. the software only consists of two files – one python file and one model file. This Python file contains a main model class which can load in the model and make predictions on entered text which it has cleaned and pre-processed. Currently, there is no user interface to interact with this prototype.

What has been tested and how?

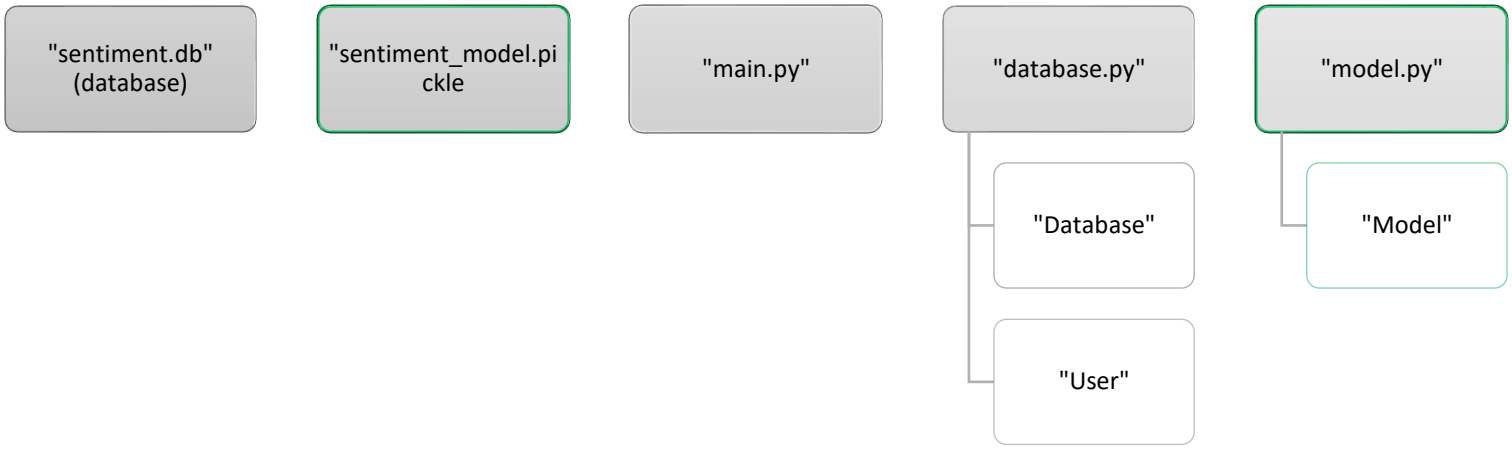
At several points through the course of the stage, I tested the Python script and the model. Testing of the model involved using a fraction of the dataset to create a test set of data. Use of this test set of data simply involved using the model to make sentiment predictions of a series of tweets and then comparing these predictions to the actual sentiment of the tweets. Its percentage of correctly predicted sentiments indicated it had an accuracy of just under 80%.

The Python script in this stage only required the use of a single class. This meant only small tests needed to be conducted. To test that the class could prepare the text and then make predictions on the text, I gave it two examples tweets.

Are there any changes that may need to be made to this component further into development?

This current prototype is using a model which currently is very simple in nature. This is primarily due to the very basic output of the model being either 1 or 0 (representing positive or negative). This means this current model does not mean the need to provide valuable predictions. To remedy, this I would need to create a completely new model which is able to provide a more detailed output. Although it would be beneficial to make this change, it is not urgent, and this current model serves the basic purpose of the model for this current protype. Moreover, due to the modularity of the current system, implementing a new model would just mean replacing the model file in the directory and shouldn’t require changes to the written code in the model file.

Current system structure (new items have a green outline)



Stage 3 – Tweet Scraper

Goals for the stage:

- Create a scraper script that can:
  - Connect to the Twitter API
  - Request a specified number of tweets relating to a specified topic

Tweet Scraper

To scrape tweets, the first thing the program needs to do is to connect to the Twitter API using my unique developer keys (I generated new keys after taking this screenshot to maintain account security and comply with the Twitter Developer Terms of Service)

```
import tweepy # Import tweepy so the program can access all of its tools

# The following constants are keys which Twitter uses to authenticate my access to their Tweets
CONSUMER_KEY = 'WqxCENBbnnzPFWFYxVmKGraE9'
CONSUMER_SECRET = 'iptXMY0NOgRJnaDXpCLjFqMsRZpOyeX5fBXekMA4DUZj8tkgge'
ACCESS_TOKEN = '961284377333428224-swaNx3JvF88Z1XYZjOrv0rRRsdzNaJC'
ACCESS_TOKEN_SECRET = 'VZsFVLJKFCL6OpzcgneKzL2XpfoFUUCpkfW2fBLefNcSm'

# The 4 keys are authenticated in a request to Twitter
auth = tweepy.OAuthHandler(CONSUMER_KEY, CONSUMER_SECRET) # Twitter authenticates the consumer keys
auth.set_access_token(ACCESS_TOKEN, ACCESS_TOKEN_SECRET) # Twitter authenticates the access keys

api = tweepy.API(auth) # Create an API object so I can access the methods I need to view tweets
```

As these keys are meant to be secret, I decided to store these in an external file that I can connect to and retrieve.

```
import tweepy
from keys import keys

# The following constants are keys which Twitter uses to authenticate my access to their Tweets
CONSUMER_KEY = keys['CONSUMER_KEY']
CONSUMER_SECRET = keys['CONSUMER_SECRET']
ACCESS_TOKEN = keys['ACCESS_TOKEN']
ACCESS_TOKEN_SECRET = keys['ACCESS_TOKEN_SECRET']

# The 4 keys are authenticated in a request to Twitter
auth = tweepy.OAuthHandler(CONSUMER_KEY, CONSUMER_SECRET)  # Twitter authenticates the consumer keys
auth.set_access_token(ACCESS_TOKEN, ACCESS_TOKEN_SECRET)  # Twitter authenticates the access keys

api = tweepy.API(auth)  # Create an API object so I can access the methods I need to view tweets
```

I then wrote these lines into a class to maintain the modularity of the program.

```
class Scraper:
    # The following constants are keys which Twitter uses to authenticate my access to their Tweets
    CONSUMER_KEY = keys['CONSUMER_KEY']
    CONSUMER_SECRET = keys['CONSUMER_SECRET']
    ACCESS_TOKEN = keys['ACCESS_TOKEN']
    ACCESS_TOKEN_SECRET = keys['ACCESS_TOKEN_SECRET']

    # The 4 keys are authenticated in a request to Twitter
    auth = tweepy.OAuthHandler(CONSUMER_KEY, CONSUMER_SECRET)  # Twitter authenticates the consumer keys
    auth.set_access_token(ACCESS_TOKEN, ACCESS_TOKEN_SECRET)  # Twitter authenticates the access keys

    api = tweepy.API(auth)  # Create an API object so I can access the methods I need to view tweets

    connection_created = False

    # If the try block produces an error, the error statement will be output
    try:
        connection_created = auth.get_authorization_url()
    except tweepy.TweepError:
        pass
```

In the above code, I also added a try-except block to notify me if there was successful connection to the API. I added this code that connects to the API outside of the constructor method ("__init__" method) so that the application will only ever connect to the API once. This is necessary as I don't want to have to instantiate the class every time I want to retrieve tweets.

To test this code worked, I ran two test cases involving the provided user access tokens. The input was in the form (consumer key, consumer secret, access token, access token secret). I also ran a test to see what would happen if my computer was not connected to the internet while trying to access the API.

Test case	Provided arguments	Expected output	Actual Output	Pass/Fail
Correct access tokens	(See screenshot above with visible access tokens)	"Successfully connected to the Twitter API"	`Successfully connected to the Twitter API`	Pass
Incorrect access tokens	"qwerty", "qwerty", "qwerty", "qwerty"	"Failed to connect to the Twitter API"	`Failed to connect to Twitter API`	Pass
Not connected to internet	N/A	"Failed to connect to the Twitter API"	`Failed to connect to Twitter API`	Pass

This showed the class to be working successfully as intended thus far.

I then added a new method to the class called "get_tweets" which uses the API to request the tweets based on the topic name and number of tweets provided when the class is instantiated. This method also uses a try-except block to handle the system not being able to retrieve the tweets for whatever reason. The method takes the topic to be search for as well as the number of tweets to search for as a parameter but by default it will attempt to find 150 tweets

```
@classmethod
def get_tweets(cls, topic_name, num_of_tweets=150):
    try:
        # Retrieve the tweets from the API
        scraped_tweets = tweepy.Cursor(cls.api.search, q=topic_name,
                                       lang='en', tweet_mode='extended', include_rts=False).items(num_of_tweets)

        # Storing the author, text, likes of the tweets in a list
        tweets_list = [[tweet.author.screen_name, tweet.full_text, tweet.favorite_count] for tweet in scraped_tweets]

        return tweets_list

    except BaseException as e:
        return []
```

Using list comprehension, the method creates a list of tweets that have been retrieved from online. It makes sure that the tweets retrieved are not retweets to avoid cluttering of the data. Using a list allows me to iterate over the tweets and perform actions on them such as outputting or cleaning them. This will also be useful when I need to sequentially give tweets to the model for it to make predictions. The list of tweets returned is a 2-Dimensional list structured in the following way:

[[Tweet Author, Tweet Text, Tweet's number of likes]]

To test this worked, I used the code below to try and get tweets about the company "Intel".

```
tweets = Scraper.get_tweets("Intel")

print(tweets)

print(f"Number of tweets retrieved: {len(tweets)}")
```

Now, when the file is run, this is the output (The outputted list of tweets is very long so I cut part of it off):

```
Successfully connected to the Twitter API
[['Haroon_Traders', '🔥HP Probook 450 G1 (
Number of tweets retrieved: 150
```

This showed that the scraper could successfully by default scrape 150 tweets if there is not a number of tweets argument provided. More importantly, it showed the scraper could successfully retrieve the tweets and the number of likes each tweet.

If I now change the code to specify the number of tweets (as indicated by the red box):

```
tweets = Scraper.get_tweets("Intel", 200)

print(tweets)

print(f"Number of tweets retrieved: {len(tweets)}")
```

The output becomes:

```
Successfully connected to the Twitter API
[['SibongileMaz', 'Your failure is also y
Number of tweets retrieved: 200
```

This small test shows that the scraper can successfully retrieve tweets from online. It also shows that by default it will retrieve 150 tweets, but this amount can be adjusted by providing relevant arguments when calling the function.

Now I will test if the model class I created earlier can return predictions on this list of scraped tweets.

To do this, I will run the following code in the main file (“main.py”):

```
import model
import scraper


tweets = scraper.Scraper.get_tweets("Intel")
print(tweets)

predictions = model.Model.make_tweet_predictions(tweets)
print(predictions)
```

Assuming this works, the program should scrape a list of 150 tweets about “Intel” and hold these tweets in a 2-Dimensional list and then make predictions on these tweets using the methods in the existing model class created in the first stage.

This was the output when run, showing the predictions on the first 5 scraped:

```
[0, 0, 0, 0, 0,
```

This showed that I can use the scraper and model in conjunction to scrape tweets and produce a list of tweets predictions.

Stage 3 – Reflection

What has been completed?

In this stage, I created a scraper able to connect to the Twitter API to scrape tweets. This involves a single class that when in imported in the main file connects to the Twitter API through Tweepy. When the class is instantiated is provided with a topic name and a number indicating the amount of tweets to be scraped.

Overview of the current prototype as a whole

Creation of the scraper in this section now means I have all the necessary functionality for the system to work. This means the primary logic and of the system is almost complete, now I need to tie the files together and implement each of the features with a user interface.

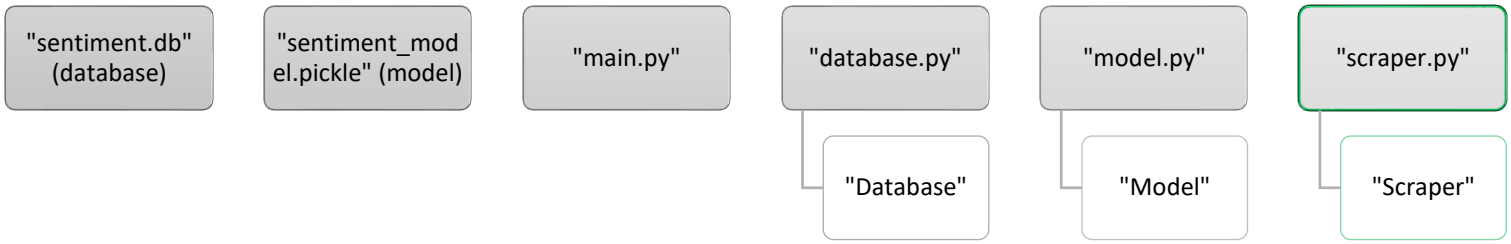
What has been tested and how?

When starting to create the scraper, I ran a small test to see if the scraper could successfully connect to the Twitter API. During this test, I also tested how the system would handle unsuccessful connection attempts. This included a connection attempt with incorrect access tokens as well as a connection attempt while not being connected to the internet. Currently, when encountering an unsuccessful connection attempt, the system will only be able to print an appropriate error message, but further into development, this will become integrated with the user interface.

Links to the success criteria

- *"The user will be able to input a topic that they want to find the public's opinions on."* – This is now possible when providing the "num_of_tweets" parameter when calling the "get_tweets" function.

Current system structure (new items have a green outline)



Stage 4 – User data handling

Now that the database, model and scraper are complete, I can start working on combining these tools in a centralised point for data handling.

First, I will create the classes that represent the user's topics and company. I intend to create a superclass for topics and the user's company and use inheritance due to the close similarities in the way the system handles their data.

```
class TopicHandler:
    def __init__(self, id, name):
        # Topic info attributes
        self.id = id
        self.name = name

        # Topic tweets attributes
        self.tweets = []
        self.likes = []
        self.authors = []

        # Data from current analysis
        self.predictions = []
        self.sentiments = []
        self.posTweets = 0
        self.negTweets = 0
        self.currentSentiment = 0.5

        # Log data
        self.historicalAverageSentiment = 0.5
        self.monthsTweets = [0 for i in range(8)]
        self.monthsAverageSentiments = [0.5 for i in range(8)]
        self.monthsAveragePosNegTweets = [0 for i in range(8)]
        self.lastWeeksTweets = [0 for i in range(7)]
        self.lastWeeksAverageSentiments = [0.5 for i in range(7)]
        self.lastWeeksAveragePosNegTweets = [0 for i in range(7)]

    def make_log(self):
        Database.make_log(self)

    def get_logs(self):

        # Use the database to get the logs on this topic/company
        logs = Database.get_logs(self)

        self.get_last_8_months(logs)
        self.get_last_week_data(logs)


    def get_last_8_months(self, logs):

        date = datetime.now().strftime("%d/%m/%y").split("/")

        # Get stats from the last 8 months
        for i in range(0, 8):
            month = (datetime.now() - timedelta(days=30*i)).strftime("%m/%Y")

            # Calculate the average sentiment this month
            sentiments_that_month = [log[0] for log in logs if datetime.strptime(log[1], "%d/%m/%Y").strftime("%m/%Y")
                                     == month]
            if len(sentiments_that_month) != 0:
                month_average_sentiment = round(sum(sentiments_that_month) / len(sentiments_that_month), 2)
                self.monthsAverageSentiments[i] = month_average_sentiment
            else:
                self.monthsAverageSentiments[i] = 0.5

            # Calculate the average number of positive and negative tweets this month
            pos_this_month = [log[2] for log in logs if datetime.strptime(log[1], "%d/%m/%Y").strftime("%m/%Y")
                              == month]
            neg_this_month = [log[3] for log in logs if datetime.strptime(log[1], "%d/%m/%Y").strftime("%m/%Y")
                              == month]

            average_pos_this_month = average_neg_this_month = 0
            if len(pos_this_month) != 0:
                average_pos_this_month = round(sum(pos_this_month) / len(pos_this_month))
            if len(neg_this_month) != 0:
                average_neg_this_month = round(sum(neg_this_month) / len(neg_this_month))

            self.monthsAveragePosNegTweets[i] = [average_pos_this_month, average_neg_this_month]

        self.monthsTweets = [sum(month) for month in self.monthsAveragePosNegTweets]

        if len(logs) != 0:
            self.historicalAverageSentiment = round(
                sum(self.monthsAverageSentiments) / len(self.monthsAverageSentiments),
                2)

    def get_last_week_data(self, logs):
        # Get stats from the last 7 days
        last_week_dates = [(datetime.now() - timedelta(days=i)).strftime("%d/%m/%Y") for i in range(7)]

        for i in range(7):
            day = last_week_dates[i]

            # Find the average sentiment this day

            days_sentiments = [log[0] for log in logs if log[1] == day]
            if len(days_sentiments) != 0:
                days_average_sentiment = round(sum(days_sentiments) / len(days_sentiments), 2)
                self.lastWeeksAverageSentiments[i] = days_average_sentiment
            else:
                self.lastWeeksAverageSentiments[i] = 0.5


            # Calculate the average number of positive and negative tweets on this day
            pos_tweets_this_day = [log[2] for log in logs if log[1] == day]
            neg_tweets_this_day = [log[3] for log in logs if log[1] == day]

            average_pos_this_day = average_neg_this_day = 0
            if len(pos_tweets_this_day) != 0:
                average_pos_this_day = round(sum(pos_tweets_this_day) / len(pos_tweets_this_day))
            if len(neg_tweets_this_day) != 0:
                average_neg_this_day = round(sum(neg_tweets_this_day) / len(neg_tweets_this_day))
            self.lastWeeksAveragePosNegTweets[i] = [average_pos_this_day, average_neg_this_day]

        self.lastWeeksTweets = [sum(day) for day in self.lastWeeksAveragePosNegTweets]
```

In brief, currently the class makes use of the logging functionality provided by the “Database” class to make and retrieve data logs. This class will be initially instantiated when the program is run and the “retrieve_logs” method will be run to access the database using the “Database” class and retrieve all of the records that refer to the instance’s name i.e the user’s company. Upon retrieving the logs about the company (for example), it breaks down the data creating lists representing data fluctuations over the last 7 days and the last 8 months. This data manipulation is handled here as it means data is ready be used and presented later when developing the user interface. These lists include a list representing the average number of tweets about the topic/company per month/day as well as other lists showing the polarity of the tweets that month/day.

The class also needs methods for making for actually performing the analysis so methods will be needed for retrieving tweets using the “Scraper” class and then using the “Model” class to make the predictions on these tweets.

```
def get_tweets(self):
    # Get the tweets, tweet authors and likes of the tweets
    self.tweets = scraper.Scraper.get_tweets(self.name)

    self.tweets = [tweet for tweet in self.tweets]

    self.authors = [tweet[0] for tweet in self.tweets]

    self.likes = [tweet[2] for tweet in self.tweets]

    self.tweets = list(set([tweet[1] for tweet in self.tweets]))

def get_predictions(self):
    # Get predictions
    self.predictions, self.sentiments = Model.make_tweet_predictions(self.tweets)
    # Current average sentiment of the most recent analysis
    if len(self.predictions) != 0:
        self.currentSentiment = round(sum(self.predictions) / len(self.predictions), 2)

    # Get the number of positive and negative tweets
    self.posTweets = self.sentiments.count("Positive")
    self.negTweets = self.sentiments.count("Negative")
```

The “get_tweets” method simply retrieves the tweets using the “Database” class and then separates the tweets into lists of text, authors and likes so they can be used later.

The “get_predictions” method uses the “Model” class to make predictions on the tweets found using the “get_tweets” method. It then calculates how many positive and negative tweets there are and also finds this average sentiment of these tweets.

```
def perform_analysis(self):
    # Get the tweets and perform analysis on these tweets
    self.get_tweets()
    self.get_predictions()
```

I have also added this method so that when wanting new analyses I can simply call this one method instead of the two created prior.

This means the “TopicHandler” class is now complete and I can begin creating the classes for topics and the user company which will inherit this class.

Below the “Company” and “Topic” classes inherit the “TopicHandler” class. Inheritance is used here to reduce the amount of repeated code, as the companies and topics will have data presented about them in very similar ways

```
class Company(TopicHandler):
    def __init__(self, companyID, companyName):
        self.type = "company"
        super(Company, self).__init__(companyID, companyName)
        self.perform_analysis()
        Database.make_log(self)
        self.get_logs()


class Topic(TopicHandler):
    def __init__(self, topicID, topic):
        self.type = "topic"
        super(Topic, self).__init__(topicID, topic)
        self.perform_analysis()
        self.make_log(self)
        self.get_logs()
```

When these two classes are instantiated, they take the name and id of the company/topic as parameters and then perform sentiment analysis, making a log of this new sentiment data, then retrieving all of the past logs of the topic/company so they can be shown on the UI (when created).

Now, I will create the class which will represent the user.

```python
class User:
    def __init__(self, username):
        self.username = username
        self.id = None
        self.company = None
        self.topics = None

        self.get_info()

    def get_info(self):
        # Get the user's information including their topics and what company they belong to.
        self.id, company, companyID = Database.get_user_info(self.username)

        # Create the user's list of topics
        self.topics = [Topic(topic[0], topic[1]) for topic in Database.get_user_topics(self.id)]

        # Create a Company instance
        self.company = Company(companyID, company)

    def new_topic(self, topicName):
        # Add the new topic to the database
        topicID = Database.new_topic(self.id, topicName)

        # Create a new Topic instance
        topic = Topic(topicID, topicName)

        # Add this topic to the user's topic list
        self.topics.append(topic)

    def remove_topic(self, topic):
        # Remove the relationship between the user and the topic
        Database.remove_topic(self.id, topic.id)

        # Remove the topic from the user's list of topics
        self.topics.remove(topic)

    def change_company(self, new_company_name):
        Database.change_company(self.username, new_company_name)
```

This class doesn't introduce any new functionality as it is primarily using existing methods in other classes to act as a central point for data handling. By using this class, the application will have access to all of the data it could need for the user interface.

The "get_info" method is initially run to retrieve the necessary information about the user such as what topics they want analysed as well as what company they belong to. It does this by using the relevant method from the "Database" class. The method then iterates

It then instantiates the "Company" class, creating a company attribute for the user.

I have also added some methods for making changes to the user's related topics stored in the database. The "new_topic" method takes the name of the topic to be added as a parameter and then uses the "Database" method created earlier to add this to the database and update all of the tables as needed. Another method is "remove_topic" which simply uses the other class to update the tables so that the user is no longer linked to that topic and then update the user's "topics" list attribute so the system know to no longer analyse this topic. Lastly is the "change_company" method which simply updates the necessary tables so that the user is linked to different company within the database.

Stage 4 – Reflection

What has been completed?

In this stage, I combined the functionality created in the previous stages in development to create a centralised point of data access. This involved creating classes representing the user's company (called "Company" and all of the user's topics (called "Topic"). These classes both inherited a superclass which provided the functionality to:

- Use the scraper to find tweets about that topic/company instance.
- Use the model to make sentiment predictions on these tweets
- Use the "Database" class to log the results of this analysis in the database
- Retrieve any logs that have been made previously about the topic/company

I then created a user class which will be used to represent the user once they have logged in. This class instantiates the "Topic" class creating a list attribute containing all of their topics to be analysed. It also instantiates the "Company" to create a company attribute. Creating the "User" class to make use of these other object types means that by using one "User" instance, I can access all of their company as well as all of their topics and access all of the topics' company's data from previous logs and results from the most recent analyses including the most recent group of tweets found talking about the company/topics. The "User" class also has some more minor functionality like the removal of topics and changing of what company they are associated with.

Overview of the current prototype as a whole

The logic and functionality of the final application is mostly complete. Now, I just need to work on integration and data embedding within a database.

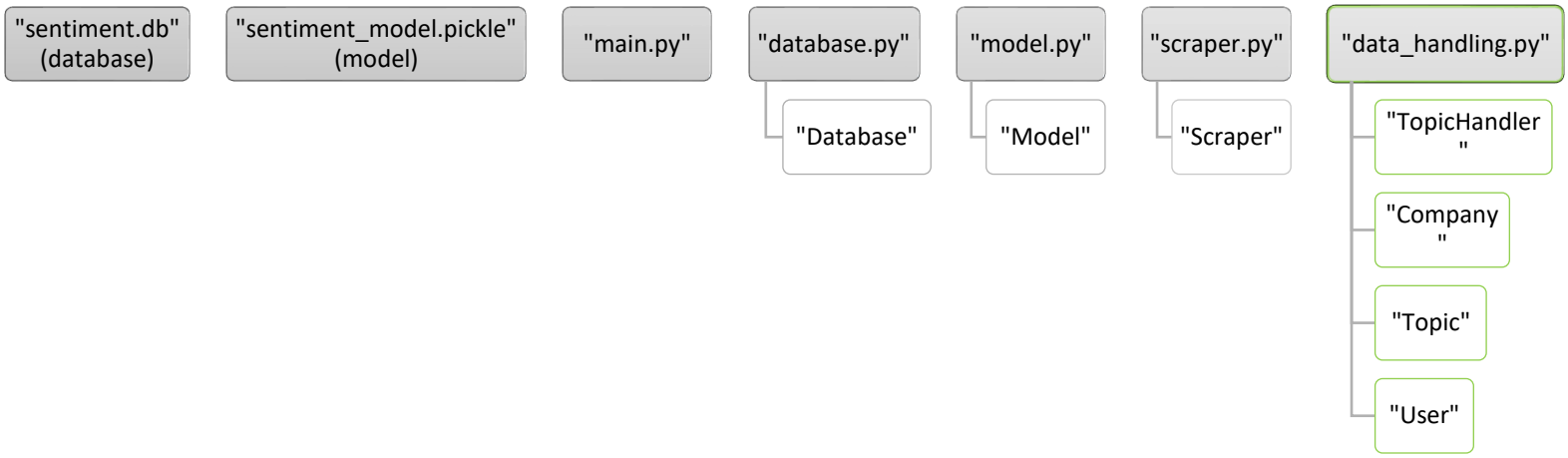
What has been tested and how?

Testing in this section was important as this area of the program combines a lot of complex functionality.

Links to the success criteria

There weren't any specific criterion addressed in this section as the functionality created is to act as a foundation for further implementation.

Current system structure (new items have a green outline)



Stage 5 – User Interface

For the user interface, I will be using the PyQt5 python library. I will need to install this using conda as before.

Once the library is installed, I can now create the classes required to provide the signup and login screens.

I first need to create a main controller class where the which controls what is currently shown by the user interface. This will go in the main file ("main.py") whereas the classes used for the user interface will be written in a new python file called "gui.py".

The following structure outlines the structure of the controller class within "main.py" which will control what is show on screen.

```
class Controller:  # Controls what is shown on the user interface
    def __init__(self):  # Constructor method
        self.widget = QtWidgets.QStackedWidget()  # Create the window stack

    def show_login(self):  # Is run to show the login screen
        pass

    def show_sign_up(self):  # Is run to show the signing up screen
        pass

    def show_main(self):  # Is run to show the main window
        pass


def gui():
    app = QApplication(sys.argv)  # Starts the application
    userInterface = Controller()  # Create an instance of the controller
    app.exec_()  # Creates a loop, displaying the application gui until it is exited
```

In this class' constructor method, I have used the "QStackedWidget()" function of PyQt5 which allows me to have the software jump between different screens depending on what part of the software the user is currently using. The stacked widget uses indexing with each form having its own index e.g a stack index of 1 can be selected when wanting to show the login form and a stack index of 3 can be used when needing to show the signup form. Both the login and sign-up classes will inherit the "QDialog" form type provided by PyQt5. This is so these forms can be placed on the stacked widget to switch from one to the other.

The controller class has 3 methods, one for showing the login form, one for showing the signing up form and one for showing the main window. In each of these methods, I will use the indexing described above to show the necessary window.

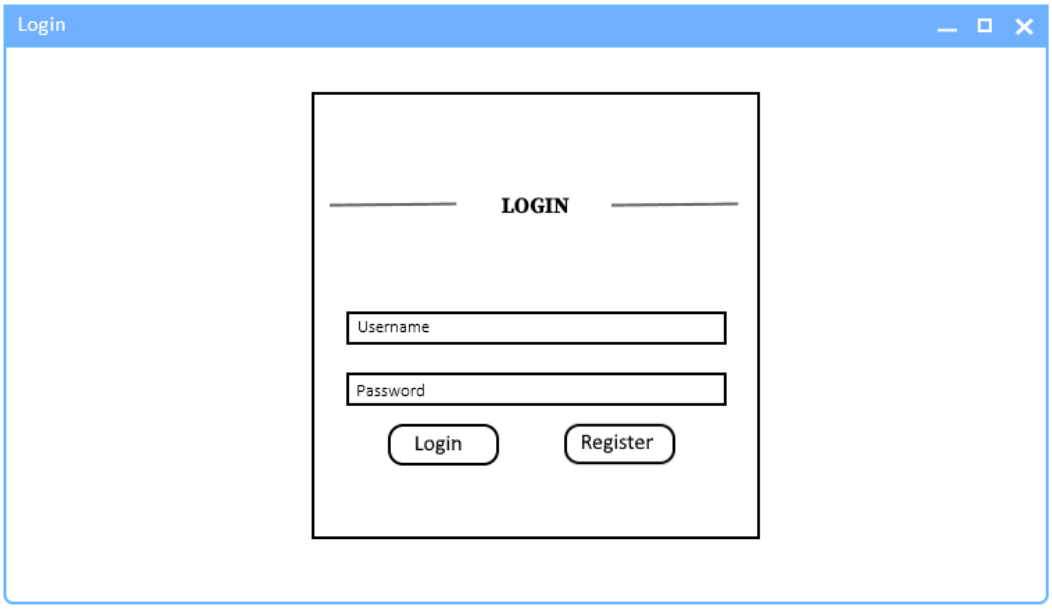
The "gui" function is what initially creates the application, instantiating the controller class to begin showing the application. The "app" variable allows the program to have an ongoing main loop which will allows for a dynamic interface.

I have also written the function below so I can get the width and height of the screen so I can appropriately configure the sizes of the window. This is so I do not have disproportionately sized windows and will also help align the windows in the centre of the screen.

```
def get_screen_size():
    sizeObject = QtWidgets.QDesktopWidget().screenGeometry(-1)
    return sizeObject.width(), sizeObject.height()
```

Now I have the controller class and can use this screen size function, I can create the user interface for the login, signup, and main window screens based on this design created earlier.

Login screen



Above is the design for the login screen used earlier that I will be using to create the login form which I will create using the Login class which I have begun constructing below.

```
class Login(QDialog):# Inherits QDialog so the window can be put on the stack widget
    def __init__(self, parent):# The parent parameter is the stack widget
        super(Login, self).__init__()
        self.parent = parent# Gives the login instance access to change the stack widget

    def initUI(self):# Where the user interface elements, e.g buttons, are created
        pass

    def login(self):# This is what is run when the login button is pressed
        print("logged in")

    def signUp(self):# This is what is run when the signup button is pressed
        print("now sign up")
```

The “initUI” method will be where I add the user input boxes and buttons to the form. The “login” and “signup” methods have been added to be used when the login and signup buttons on the form are used.

I have now also updated the controller class to add the login screen to the stacked widget using the “addWidget” function.

```
class Controller:  # Controls what is shown on the user interface
    def __init__(self):  # Constructor method
        self.widget = QtWidgets.QStackedWidget()  # Create the window stack

        self.login = Login(self)  # Instantiates the login screen
        self.widget.addWidget(self.login)  # Adds the login to the stack

        self.show_login()  # Runs the "show_login" method, showing the login

    def show_login(self):  # Is run to show the login screen
        self.login.initUI()  # Sets up the user interface elements of the login screen
        self.widget.setCurrentIndex(0)  # Sets the index of the stack so the login screen is showed
        self.widget.show()  # Shows the updated stack (i.e shows the login screen)
```

In the constructor method, I have created the widget stack and then added to the login form to this stack. As I want the login screen to be the first thing shown on screen, I have also used the “show_login” method to be initially run. This method initialises the graphics required for the login form and then adjusts the current index on the widget stack so this form is shown. It then uses the “show()” function to show the current item on the stack, i.e the login form.

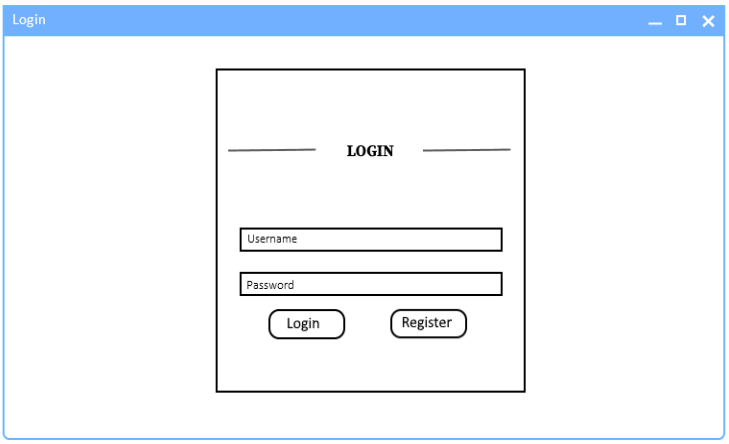
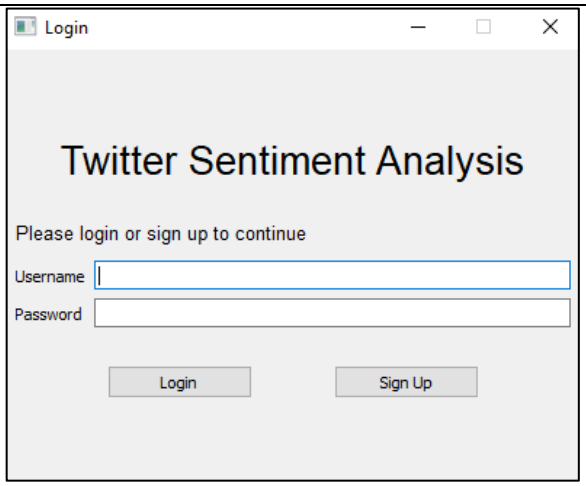
Below is the finished “initUI” method for the login screen.

Initially, I set the geometry of the window i.e. the dimensions of the window and where the window goes on the screen.

I then create a main layout for the window called “windowLayout” which contains everything needed for the form. Within this layout, I have smaller layouts which include; the Title for the window; a prompt asking the user to enter their username and password; the input fields for the user to enter their username and password; an error message which is initially hidden and then shown if the user enters incorrect login details; a “login” button for the user to submit their details and a “sign up” button if the user does not yet have an account (These two buttons have an associated function which is run when the buttons are clicked which can be seen in the initial screenshot of the “Login” class). I then add each of these layouts to the main window layout and add spacings of varying sizes between them for aesthetic purposes. Lastly, I then give the window this main window layout which now contains all the necessary components.

```
class Login(QDialog):  # Inherits QDialog so the window can be put on the stack widget
    def __init__(self, parent):  # The parent parameter is the stack widget
        super(Login, self).__init__()
        self.parent = parent  # Gives the login instance access to change the stack widget

    def initUI(self):  # Initialise the UI
        # Change the window title
        self.parent.widget.setWindowTitle("Login")

        # Establish geometry and centre the window
        width = 400
        height = int(width * 3 / 4)
        self.parent.widget.setGeometry(get_screen_size()[0] / 2 - width / 2, get_screen_size()[1] / 2 - height / 2, 0,
                                       0)

        # Set fixed dimensions for the window so the user can't accidentally change the window size
        self.parent.widget.setFixedSize(width, height)

        # Create layout for the window
        windowLayout = QVBoxLayout()
        windowLayout.setAlignment(Qt.AlignCenter)

        # Create layouts to go on the window:
        # Tile Layout
        titleLayout = QHBoxLayout()
        titleLayout.setAlignment(Qt.AlignHCenter)
        title = QtWidgets.QLabel("Twitter Sentiment Analysis")
        title.setFont(QFont("Arial", 20))
        titleLayout.addWidget(title)

        prompt = QtWidgets.QLabel("Please login or sign up to continue")
        prompt.setFont(QFont("Arial", 10))

        # Fields layout
        fieldsLayout = QFormLayout()
        self.username = QLineEdit()
        self.password = QLineEdit()
        self.password.setEchoMode(QtWidgets.QLineEdit.Password)
        fieldsLayout.addRow("Username", self.username)
        fieldsLayout.addRow("Password", self.password)

        # Error message layout
        self.errorMsg = QLabel("Incorrect Username or password")
        self.errorMsg.setAlignment(Qt.AlignCenter)
        self.errorMsg.setStyleSheet('color: red')
        self.errorMsg.hide()

        # Button layout
        buttonLayout = QHBoxLayout()
        buttonLayout.setAlignment(Qt.AlignCenter)

        loginButton = QPushButton("Login")
        loginButton.setFixedWidth(100)
        loginButton.clicked.connect(self.login)

        signupButton = QPushButton("Sign Up")
        signupButton.setFixedWidth(100)
        signupButton.clicked.connect(self.signUp)

        buttonLayout.addWidget(loginButton)
        buttonLayout.addSpacing(50)
        buttonLayout.addWidget(signupButton)

        # Add to elements layout
        windowLayout.addLayout(titleLayout)
        windowLayout.addSpacing(20)
        windowLayout.addWidget(prompt)
        windowLayout.addSpacing(5)
        windowLayout.addLayout(fieldsLayout)
        windowLayout.addSpacing(5)
        windowLayout.addWidget(self.errorMsg)
        windowLayout.addSpacing(10)
        windowLayout.addLayout(buttonLayout)

        # Set the window's main layout
        self.setLayout(windowLayout)
```

This is the outcome when the program is now run.

Initial design	Outcome
	

To test the buttons:

When the “Login” button is clicked, this is the output:

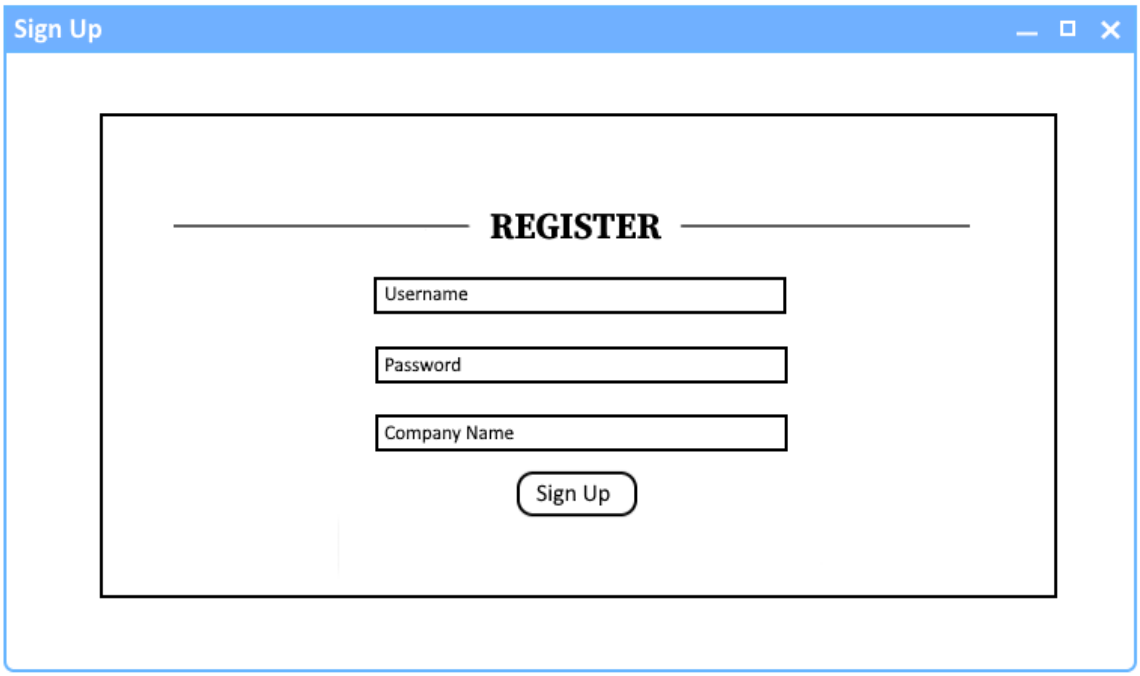
```
logged in
```

And when the “Sign Up” button is clicked:

```
now sign up
```

These two small tests show that the buttons and their assigned subroutines will work as intended but currently subroutines only contain print statements. Eventually, these subroutines will also involve graphical changes such as closing the login form and showing the main window when pressing the “login” button. These methods will also need to be integrated with the database class that I created earlier to allow for the authentication of user details when logging in.

Sign Up



The sign-up form is similar to the login screen so I can use the same code with some minor adjustments such as including another input field asking for the user's company. Another difference is the unlabelled error message. In the class for logging in, the error message simple read "Invalid username or password" but for signing up I need the error message to be able to say "Username is already taken" as well as "Invalid username or password" so I haven't defined its label yet.

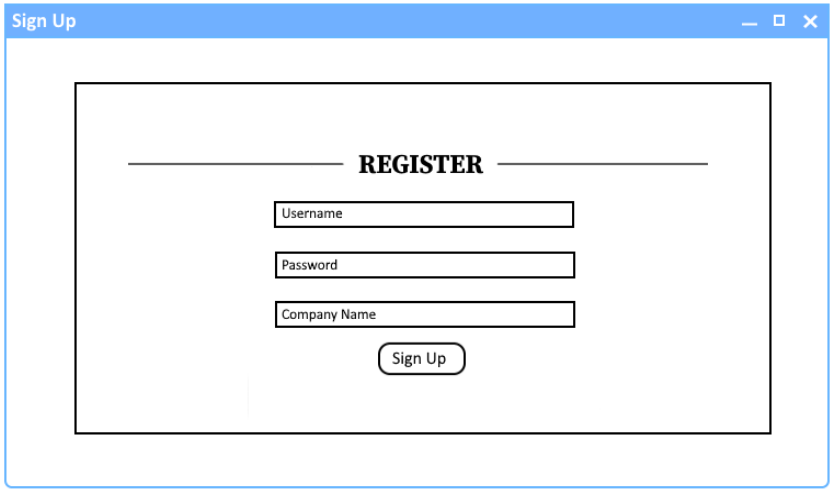
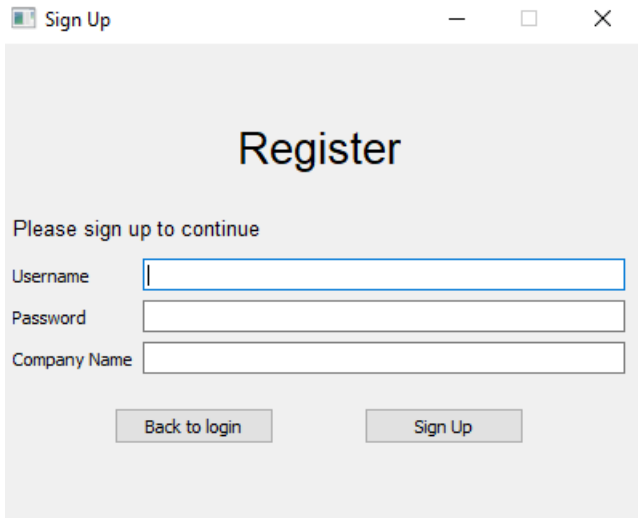
```
class SignUp(QDialog):
    def __init__(self, parent):
        super(SignUp, self).__init__()
        self.parent = parent

    def initUI(self):  # Initialise the UI
        # Change the window title
        self.parent.widget.setWindowTitle("Sign Up")

        # Establish geometry
        width = 400
        height = int(width * 3 / 4)

        # Change the width of the controller widget to be the size of the sign up screen
        self.parent.widget.setGeometry(get_screen_size()[0] / 2 - width / 2, get_screen_size()[1] / 2 - height / 2, 0,
                                       0)

        # Set fixed dimensions for the window so the user can't accidentally change the window size
        self.parent.widget.setFixedSize(width, height)

        # Create layout for the window
        windowLayout = QVBoxLayout()
        windowLayout.setAlignment(Qt.AlignCenter)

        # Create layouts to go on the window:
        # Tile Layout
        titleLayout = QHBoxLayout()
        titleLayout.setAlignment(Qt.AlignHCenter)
        title = QtWidgets.QLabel("Register")
        title.setFont(QFont("Arial", 20))
        titleLayout.addWidget(title)

        prompt = QtWidgets.QLabel("Please sign up to continue")
        prompt.setFont(QFont("Arial", 10))

        # Fields layout
        fieldsLayout = QFormLayout()
        self.username = QLineEdit()
        self.password = QLineEdit()
        self.password.setEchoMode(QtWidgets.QLineEdit.Password)
        self.company = QLineEdit()
        fieldsLayout.addRow("Username", self.username)
        fieldsLayout.addRow("Password", self.password)
        fieldsLayout.addRow("Company Name", self.company)

        # Creating the error message widget
        self.errorMsg = QLabel(self)
        self.errorMsg.setAlignment(Qt.AlignCenter)
        self.errorMsg.setStyleSheet('color: red')
        self.errorMsg.hide()

        # Button layout
        buttonLayout = QHBoxLayout()
        buttonLayout.setAlignment(Qt.AlignHCenter)

        signupButton = QPushButton("Sign Up")
        signupButton.setFixedWidth(100)
        signupButton.clicked.connect(self.signUp)

        loginButton = QPushButton("Back to login")
        loginButton.setFixedWidth(100)
        loginButton.clicked.connect(self.login)

        buttonLayout.addWidget(loginButton)
        buttonLayout.addSpacing(50)
        buttonLayout.addWidget(signupButton)

        # Add to elements layout
        windowLayout.addLayout(titleLayout)
        windowLayout.addSpacing(20)
        windowLayout.addWidget(prompt)
        windowLayout.addSpacing(5)
        windowLayout.addLayout(fieldsLayout)
        windowLayout.addSpacing(5)
        windowLayout.addWidget(self.errorMsg)
        windowLayout.addSpacing(10)
        windowLayout.addLayout(buttonLayout)

        # Set the window's main layout
        self.setLayout(windowLayout)
```

I also need to adjust the controller class, so that the sign-up form is added to the stack and can be switched to.

```
class Controller:  # Controls what is shown on the user interface
    def __init__(self):  # Constructor method
        self.widget = QtWidgets.QStackedWidget()  # Create the window stack

        self.login = Login(self)  # Instantiates the login screen
        self.widget.addWidget(self.login)  # Adds the login to the stack

        self.signup = SignUp(self)  # Instantiates the signing up screen
        self.widget.addWidget(self.signup)  # Adds the signing up to the stack

        self.show_sign_up()  # Runs the "show_sign_up" method, showing the sign up form

    def show_sign_up(self):  # Is run to show the signing up screen
        self.signup.initUI()  # Sets up the user interface elements of the sign up screen
        self.widget.setCurrentIndex(1)  # Sets the index of the stack so the sign up screen is shown
        self.widget.show()  # Shows the updated stack (i.e shows the login screen)
```

(I have adjusted the controller constructor method to have “show_sign_up()” instead of “show_login()” to test what this new form looks like.)

This is the result of creating this class for signing up. As with the

Initial design	Outcome
	

When the “Sign Up” button is pressed, this is the result:

```
signed up
```

This shows the button works. However, I still need to add the ability to make changes to the database from this form as well as the ability to switch between windows.

Login and Signup Navigation

Login form

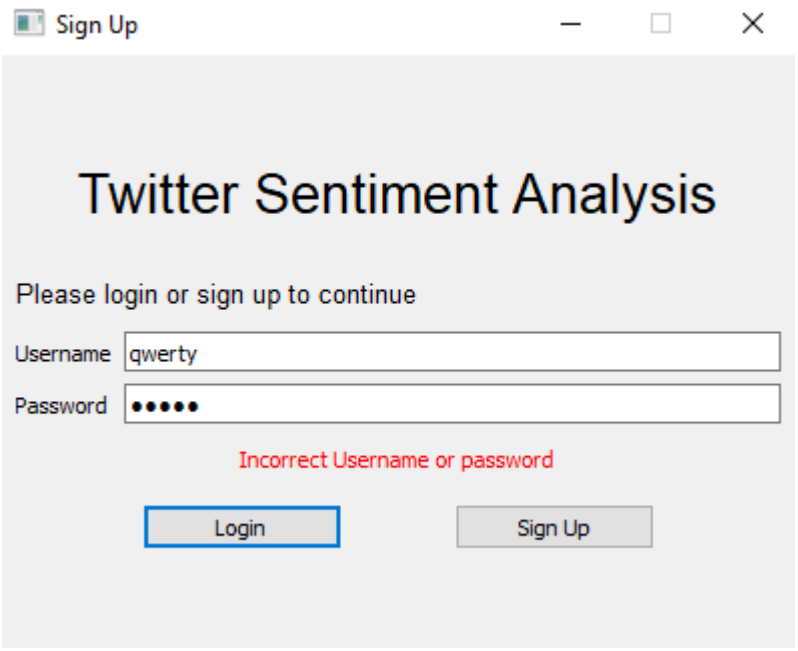
Now I have the designs for the signup and login windows, I can add the features needed for navigation.

The following is a method I have added to the “Login” class:

```
def login(self):

    # Get the entered login details
    entered_username = self.username.text()
    entered_password = self.password.text()

    # Check database to see if login details are valid
    credentials_valid = Database.authenticate(entered_username, entered_password)

    if credentials_valid:
        # Run the controller function which shows the main window
        self.parent.show_main_window()
    else:
        # Show the error message
        self.errorMsg.show()

def signUp(self):
    # Show the sign up screen
    self.parent.show_sign_up()
```

This means that now, when the login button is pressed on the login window, the system retrieves the entered login details and the validates the details using the “authenticate” method created within the “Database” class. If these login details are valid, the method within the “Controller” class that shows the main window is run. If the details are not valid, the red error message is shown on the login form. (The “signUp” method has also been added to run the “Controller” method that shows the form for signing up)

To quickly test this, I ran the program and entered invalid details and this was the result:



I will test this validation more thoroughly once the login system is completely finished.

Sign up form

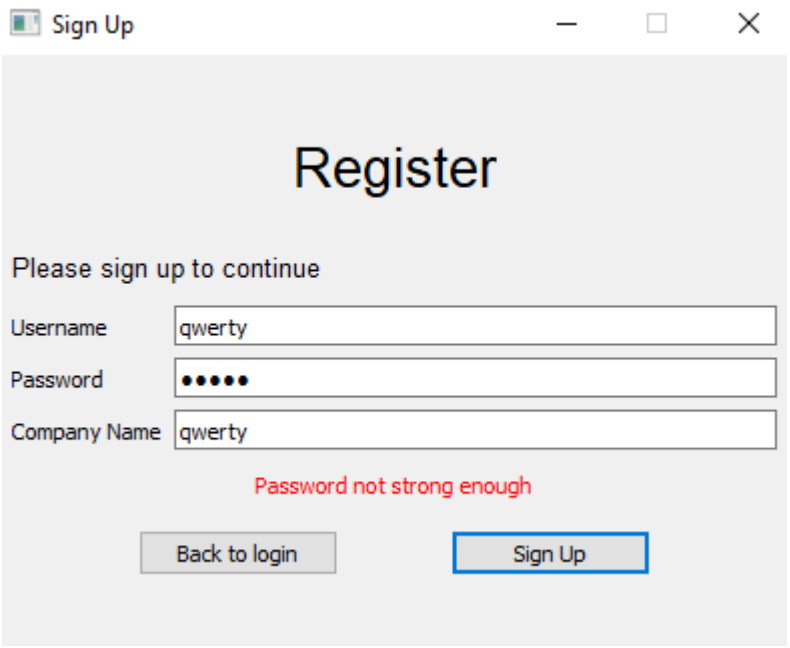
The login screen needs to be able to jump to the main window or the signing up form. The two methods below within the "Login" class allow for this. Now, when the "login" button is pressed on the login form, the username and password are authenticated using the "authenticate" function I created when making the "Database" class. If the login details are invalid, the error message I created for the window is shown. If the details are valid, the login form is closed, and the main window is shown. Also, when the "sign up" button is pressed on the login form, the login form is closed, and the signup form is shown.

Within the "Login" class:

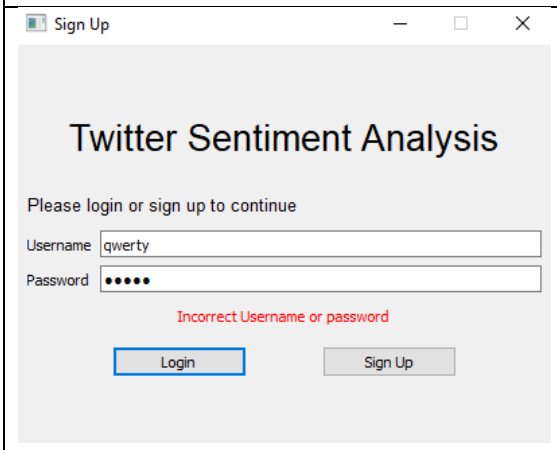
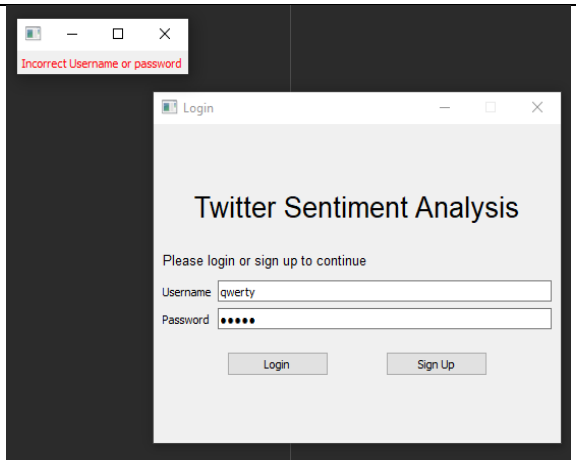
Now, I need to add this ability to switch windows to the signing up window by making similar changes to the "SignUp" class.

```
def signUp(self):
    # Get the entered details
    entered_username = self.username.text()
    entered_password = self.password.text()
    entered_company = self.company.text()

    # Check if details valid and if they are, add to database
    error = Database.new_user(entered_username, entered_password, entered_company)

    if error:
        # Show the relevant error message provided by the "new_user" function
        self.errorMsg.setText(error)
        self.errorMsg.show()
    else:
        # Run the controller function which shows the main window
        self.parent.show_main_window()

def login(self):
    # Go back to the login screen
    self.parent.show_login()
```

And when the “Sign Up” button (indicated by the red box) is pressed, the user is taken to the login screen and if they enter invalid details, the sign-up screen will become:



Here, I ran into an issue when going from the login screen to the sign-up screen then going back to login screen and entering incorrect details. Instead of being shown an error message correctly like it is above, a new window would be created showing the error like this:

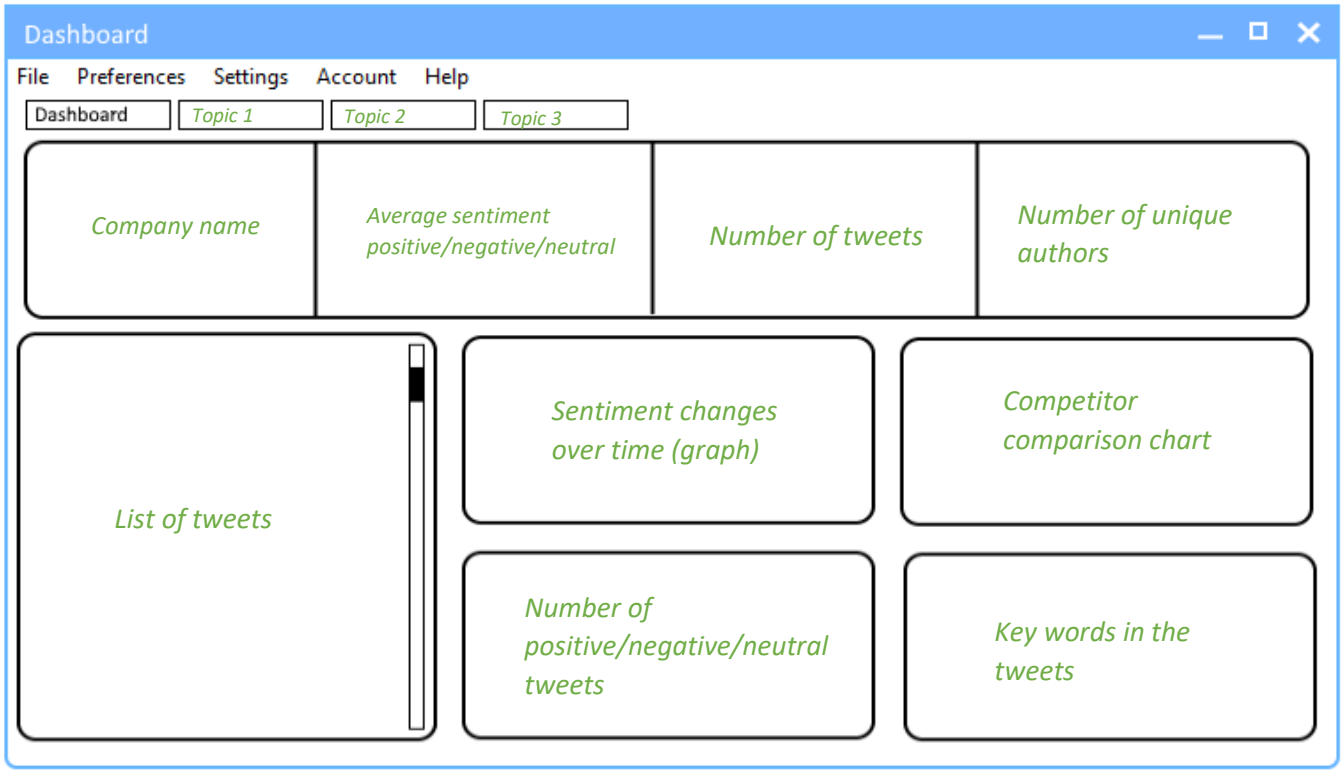
What should be shown	What is now shown
	 (Error message shown in a new window)

I realised this was due to poorly written and inefficient code I had written in the controller class which would re-initialise the screens and therefore all their buttons and text boxes each time the user switched between windows. I fixed this by only initialising each of the screens' components once at the start and then hiding the screen not currently in use, also changing the name of the stacked widget when necessary. This fixed the issue. This is a screenshot of the updated controller class in the main file:

```
class Controller:  # Controls what happens during the running of the program
    def __init__(self):
        self.widget = QtWidgets.QStackedWidget()  # Create the window stack

        self.login = gui.Login(self)  # Instantiates the login screen
        self.login.initUI()
        self.widget.addWidget(self.login)  # Adds the login to the stack

        self.signup = gui.SignUp(self)
        self.signup.initUI()
        self.widget.addWidget(self.signup)

        self.mainWindow = gui.MainWindow(self)

        self.show_login()  # Runs the "show_sign_up" method, showing the sign up form
        self.widget.show()  # Shows the current window on the stack

    def show_sign_up(self):  # Is run to show the signing up screen
        self.widget.setWindowTitle("Login")
        self.widget.setCurrentIndex(1)  # Sets the index of the stack so the sign up screen is shown

    def show_login(self):  # Is run to show the login screen
        self.widget.setWindowTitle("Sign Up")
        self.widget.setCurrentIndex(0)  # Sets the index of the stack so the login screen is shown

    def show_main_window(self, username):  # Is run to show the main window

        self.widget.hide()  # Hide the login and sign up window stack
        self.mainWindow.initUI() # Set up the user interface elements on the main window
```

REMOVE CODE IN “SHOWN_MAIN_WINDOW”

Main Window



First, I need to create the main window itself. This main window is not part of the widget stack. This is because I initially need to be able to switch between windows such as switching from the login form to the sign-up form to the main window itself, but once the main window is open, there is not a need to switch between any more windows so it can act as an individual window separate to the stack.

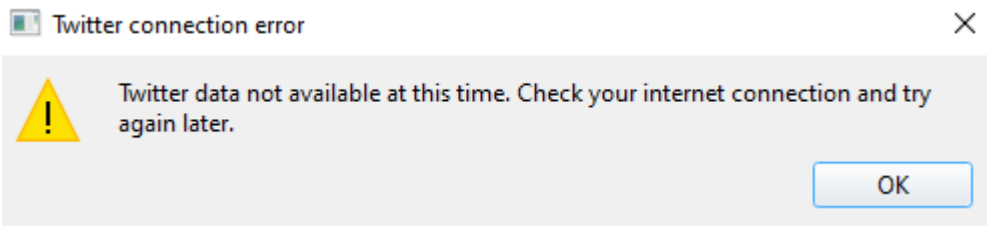
This is the initial main window class including the “add_menu” method to implement the menu at the top of the screen.

```
class MainWindow(QMainWindow):
    def __init__(self, parent, user):
        super(MainWindow, self).__init__()
        self.parent = parent

        self.user = user

    def initUI(self):
        # Set window disease
        self.setWindowTitle('Sentiment Analysis')

        # Establish geometry
        width = 1500
        height = int(width * 9 / 16)

        # Centre the window
        self.setGeometry(get_screen_size()[0] / 2 - width / 2, get_screen_size()[1] / 2 - height / 2, width, height)

        # Maximise the window
        self.showMaximized()

        # Add menu
        self.add_menu()

        # Shows the main window on the screen
        self.show()

    def add_menu(self):
        # Add the menu to the window
        mainMenu = self.menuBar()

        # File menu:
        mainMenu = mainMenu.addMenu("File")

        # Preferences menu:
        prefMenu = mainMenu.addMenu("Preferences")

        # Settings menu:
        setMenu = mainMenu.addMenu("Settings")

        # Account menu:
        accMenu = mainMenu.addMenu("Account")

        # Help menu:
        helpMenu = mainMenu.addMenu("Help")
```

I have also added a method for check the connection to the Twitter API. This will display a warning popup if the connection to the Twitter is not possible and hence a current analysis of tweets is not available. (I intend for the user to still have access to the log data despite not having access to immediate analysis)

```
def check_connection(self):
    # Check connection to Twitter
    from scraper import Scraper

    # If connection not created, show error pop up
    if not Scraper.connection_created:
        connectionPopUp = QMessageBox()
        connectionPopUp.setWindowTitle("Twitter connection error")
        connectionPopUp.setText("Twitter data not available at this time. Check your internet "
                                "connection and try again later.")
        connectionPopUp.exec_()
```

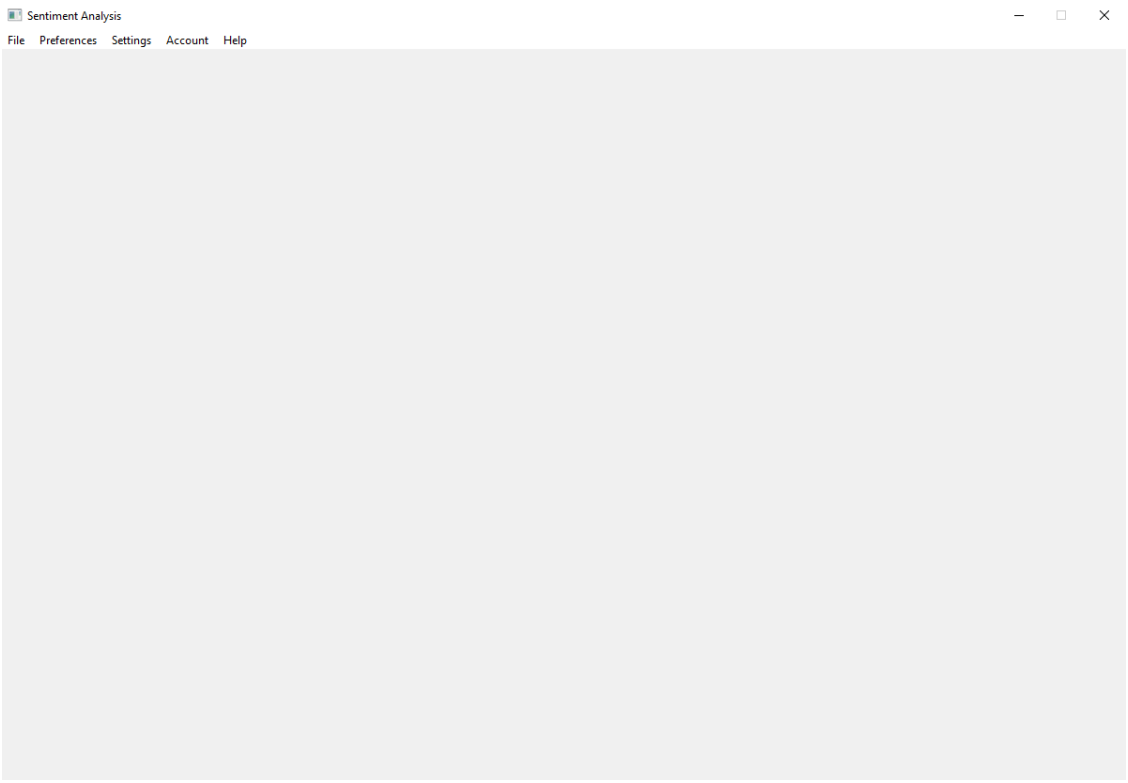
This is run immediately when the main window is shown. Showing the following popup if necessary:



Below is the updated controller function. In the constructor method, the main window is never added to the stack and once the “show_main_window” is called, the stacked widget is hidden as it is no longer needed. (the methods to show the login and sign-up forms are not shown in the screenshot)

```
class Controller:  # Controls what is shown on the user interface
    def __init__(self):  # Constructor method
        self.widget = QtWidgets.QStackedWidget()  # Create the window stack

        self.login = Login(self)  # Instantiates the login screen
        self.widget.addWidget(self.login)  # Adds the login to the stack

        self.signup = SignUp(self)  # Instantiates the signing up screen
        self.widget.addWidget(self.signup)  # Adds the signing up to the stack

        self.mainWindow = MainWindow(self)  # Instantiates the main window (don't need to add to the stack)

        self.show_main_window()  # Runs the "show_sign_up" method, showing the sign up form

    def show_main_window(self):  # Is run to show the main window
        self.widget.hide()  # Hide the widget stack
        self.mainWindow.initUI()  # Sets up the user interface elements of the main window
```

When run a simple window, titled “Sentiment Analysis” with the necessary menu bar options is displayed as shown.

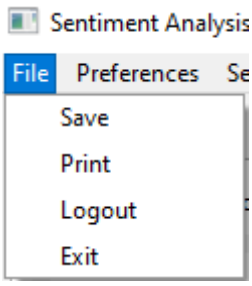


Currently each of the menu items have no functionality so I will begin to implement some of the more basic functionality now.

File menu

Now I will add the file menu and all of the file menu options:

The following code adds the necessary buttons to the file menu:



"Save":

```
def save(self):
    # Get the current date
    date = datetime.now().strftime("%d-%m-%Y")

    # Create a directory to store the screenshot if not already created
    if not os.path.exists(f"SavedData/{date}/"):
        os.makedirs(f"SavedData/{date}")

    # Get the current screen
    screen = QtWidgets.QApplication.primaryScreen()

    # Take a screenshot of the screen
    screenshot = screen.grabWindow(self.tab_widget.dashboard.winId())

    # Get the current tab
    if self.tab_widget.tabs.currentIndex() == 0:
        tab = self.tab_widget.dashboard
    else:
        tab = self.tab_widget.tabsList[self.tab_widget.tabs.currentIndex()-1]

    # Save the screenshot of the tab with the appropriatre time scale in the photo name
    if tab.timeScale == "lastWeek":
        screenshot.save(f"SavedData/{date}/{tab.name} (Last 7 Days).jpg", "jpg")
    else:
        screenshot.save(f"SavedData/{date}/{tab.name} (Last 8 Months).jpg", "jpg")
```

“Print”:

“Logout”:

This is the code required for logging out of the system. It needs to close the main window before showing the login window and then

```
def sign_out(self):
    # Close the main window
    self.close()

    # Wait for 0.25 seconds before showing the login window
    time.sleep(0.25)

    # Deleted any entered text on the login and signup windows
    self.parent.login.username.setText("")
    self.parent.login.password.setText("")
    self.parent.signup.username.setText("")
    self.parent.signup.password.setText("")
    self.parent.signup.company.setText("")
    self.parent.show_login()
    self.parent.widget.show()
```

“Exit”:

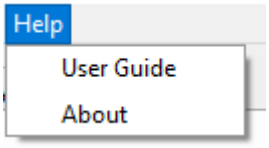
This can be done using one line so I haven't added a dedicated function, I have just asked it to close the main window when the exit button is clicked:

```
def add_menu(self):
    # Add the menu bar to the window
    mainMenu = self.menuBar()

    # File:
    fileMenu = mainMenu.addMenu('File')
    saveButton = QAction('Save', self)
    saveButton.triggered.connect(self.save)
    fileMenu.addAction(saveButton)
    printButton = QAction('Print', self)
    fileMenu.addAction(printButton)
    logoutButton = QAction('Logout', self)
    logoutButton.triggered.connect(self.sign_out)
    fileMenu.addAction(logoutButton)
    exitButton = QAction('Exit', self)
    exitButton.triggered.connect(self.close)
    fileMenu.addAction(exitButton)
```

Preferences menu

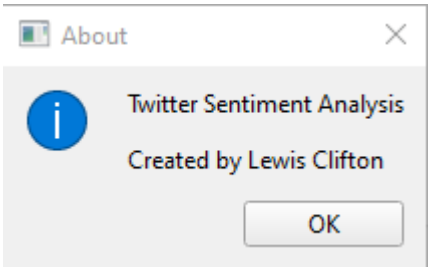
Now I will add the preferences menu

I have added a few lines that




```
def about(self):

    msg = QMessageBox()
    msg.setIcon(QMessageBox.Information)

    msg.setText("Twitter Sentiment Analysis")
    msg.setInformativeText("Created by Lewis Clifton")
    msg.setWindowTitle("About")

    _ = msg.exec_()
```

Now, when "About" is clicked, the following popup is shown:



Now, when “User Guide” is clicked, the following popup is shown:

##

This is the tab system I have added which I will explain below.

```
class TabHandler(QWidget):
    def __init__(self, parent):
        super(QWidget, self).__init__(parent)

        # Create a layout to put all of the tabs on
        self.tabLayout = QVBoxLayout(self)

        # Initialize tab screen
        self.tabs = QTabWidget()
        self.tabs.resize(300, 200)

        # Add the dashboard
        dashboard = Dashboard()
        self.tabs.addTab(dashboard, "Dashboard")

        # Add a tab
        # for i in range(10):
        self.add_tab()

        # Add all of the tabs to the tab layout
        self.tabLayout.addWidget(self.tabs)

        # Place the tab layout on the window
        self.setLayout(self.tabLayout)

    def add_tab(self):  # Add a tab
        tabName = "newtab"
        # Create the new tab
        newTab = TopicTab(tabName)
        self.tabs.addTab(newTab, tabName)


class Tab(QWidget):
    def __init__(self):
        super(Tab, self).__init__()

    def initUI(self):
        pass


class Dashboard(Tab):
    def __init__(self):
        super(Dashboard, self).__init__()
        self.name = "Company name"
        self.initUI()


class TopicTab(Tab):
    def __init__(self, name):
        super(TopicTab, self).__init__()
        self.name = name
        self.initUI()
```

The class “TabHandler” is where I can control each of the tabs from and is where I can add new tabs to a tab layout.

The “Tab” class inherits “QWidget” and is a blueprint for each tab that will go on the main window.

There are two types of tabs, the dashboard tab and user topic tabs. Looking at the designs of these two types of tab below, it is clear they are very similar with the exception of a few data elements. As they have many common features, I have decided to use a main “Tab” class with common visual elements to both the dashboard and topic tabs. This is instead of just having a large dashboard class and a large topic tab class where there would be a lot of repeated code which would be far less efficient.

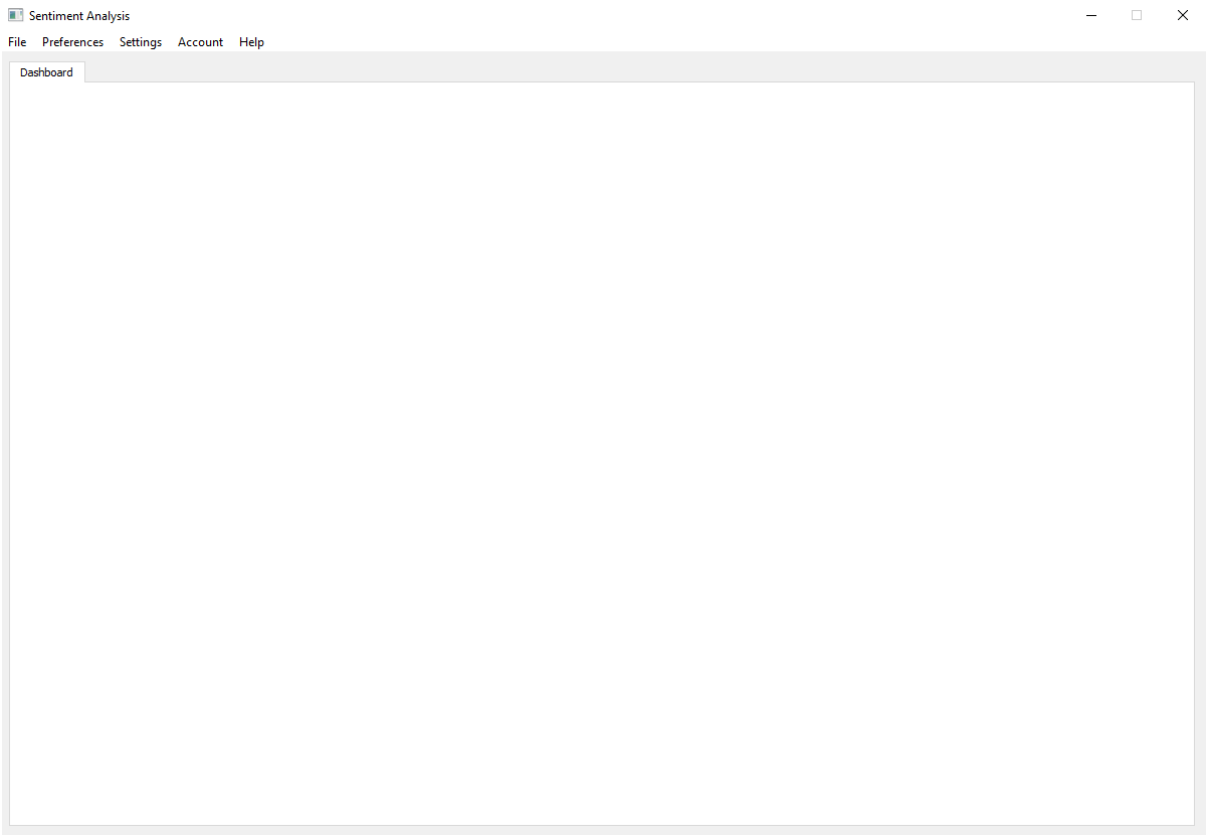
I have now updated the constructor method of the main window class so to include the tab layout with the included tabs indicated by the red box.

```
class MainWindow(QMainWindow):
    def __init__(self, parent):
        super(MainWindow, self).__init__()
        self.parent = parent

    def initUI(self):  # Initialise the UI
        self.setWindowTitle('Sentiment Analysis')

        # Establish geometry
        width = 1200
        height = int(width * 2 / 3)

        # Centre the window
        self.setGeometry(get_screen_size()[0] / 2 - width / 2, get_screen_size()[1] / 2 - height / 2, 0, 0)

        # Set fixed dimensions for the window so the user can't accidentally change the window size
        self.setFixedSize(width, height)

        # Add menu
        self.add_menu()

        # Add Tabs
        self.tab_widget = TabHandler(self)
        self.setCentralWidget(self.tab_widget)

        self.show()  # Shows the main window on the screen
```

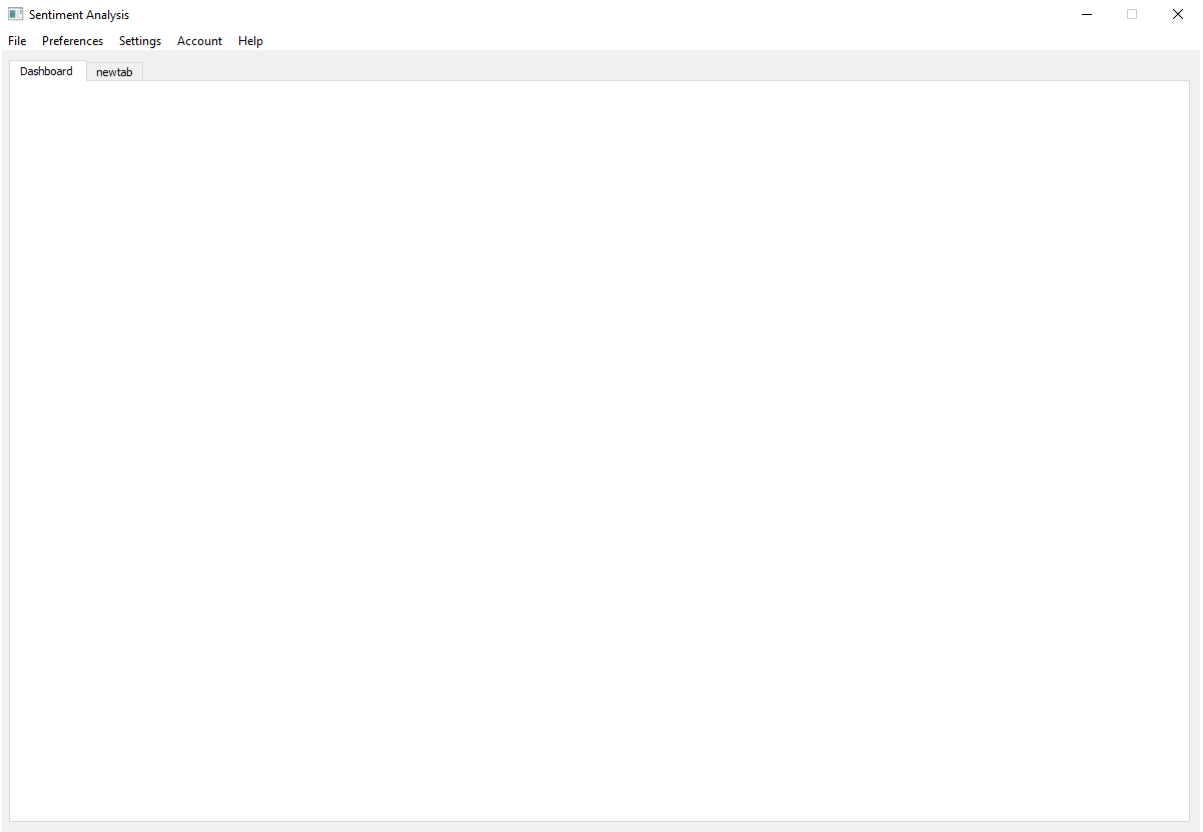
The screenshots shown below are tests of the different possible instances of tabs being used (without any visual elements which I will add later).

One dashboard tab, no topic tabs – what may be shown when the user first starts using the software:

(implemented by not calling the “add_tab” method)

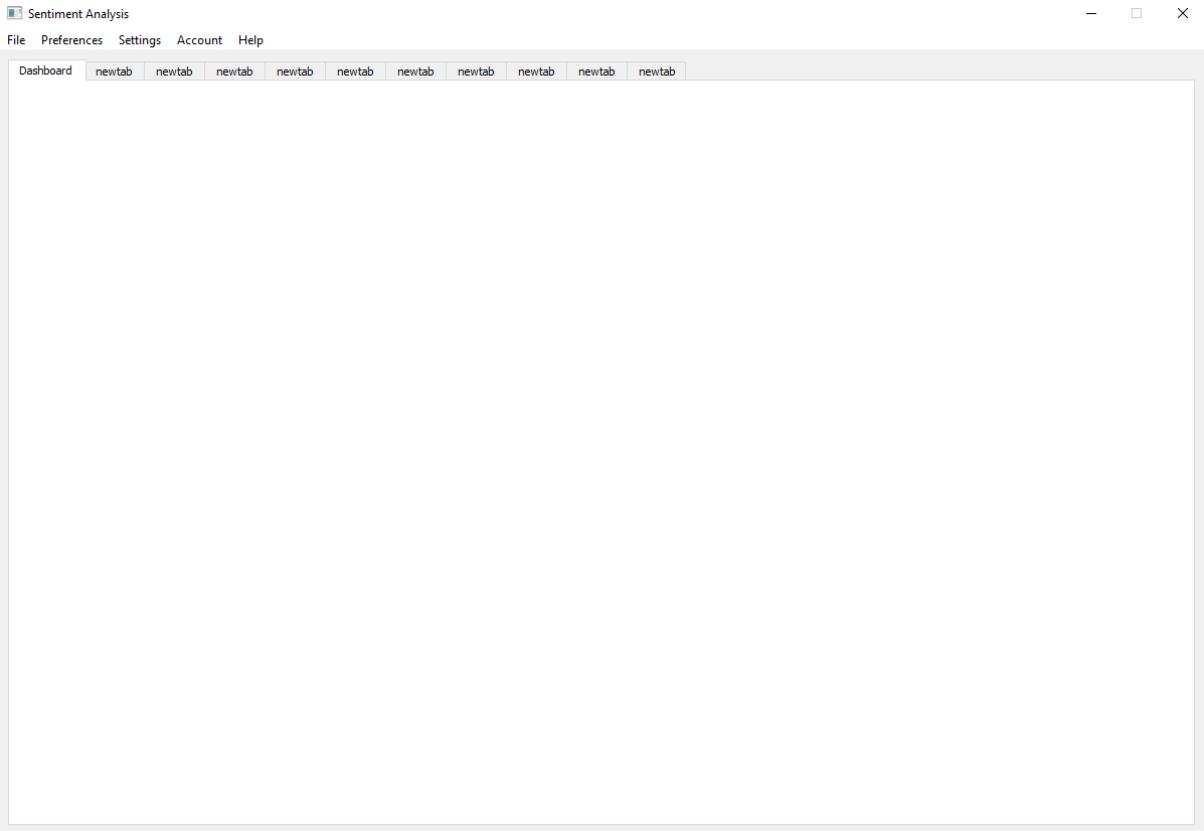


One dashboard tab, one topic tab:

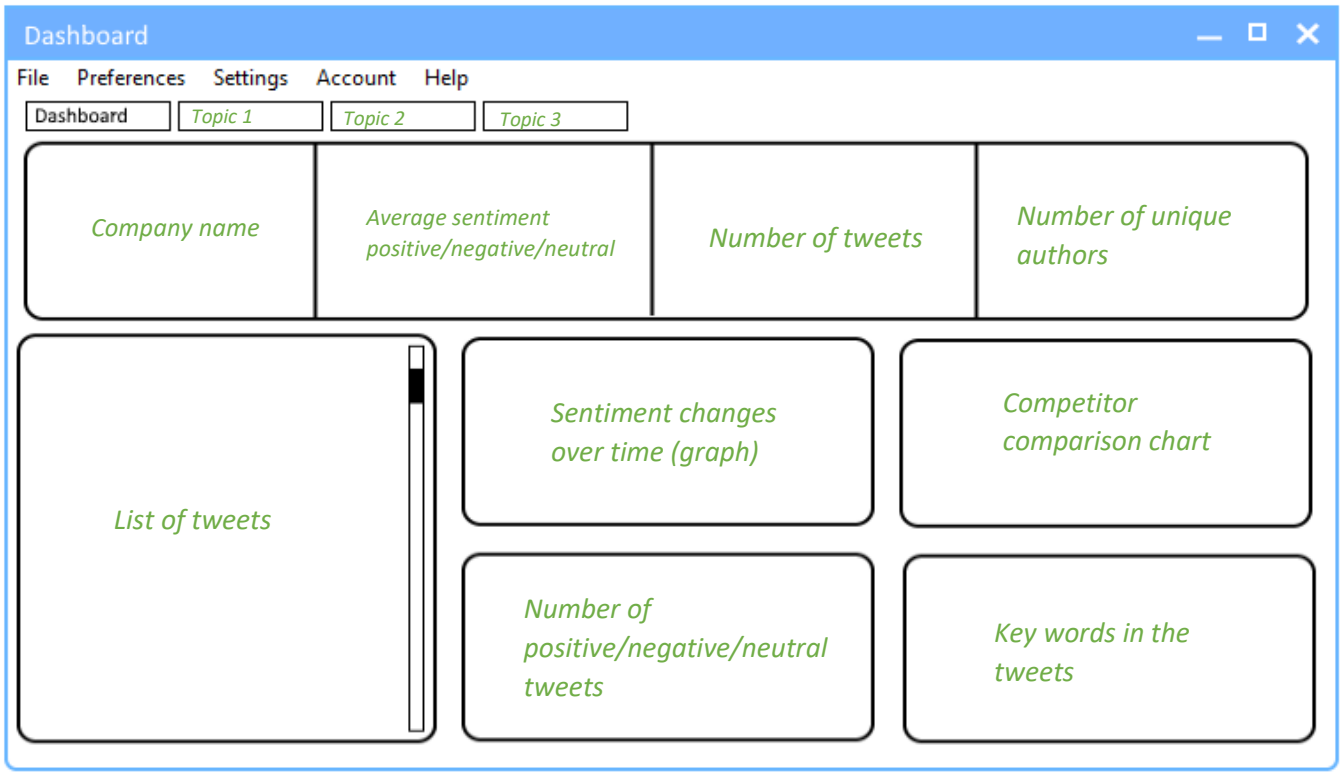


One dashboard tab, ten topic tabs – what the user may see after using the software for some time and be requiring the analysis of multiple topics:

(implemented using the “add_tab” method within a for loop)

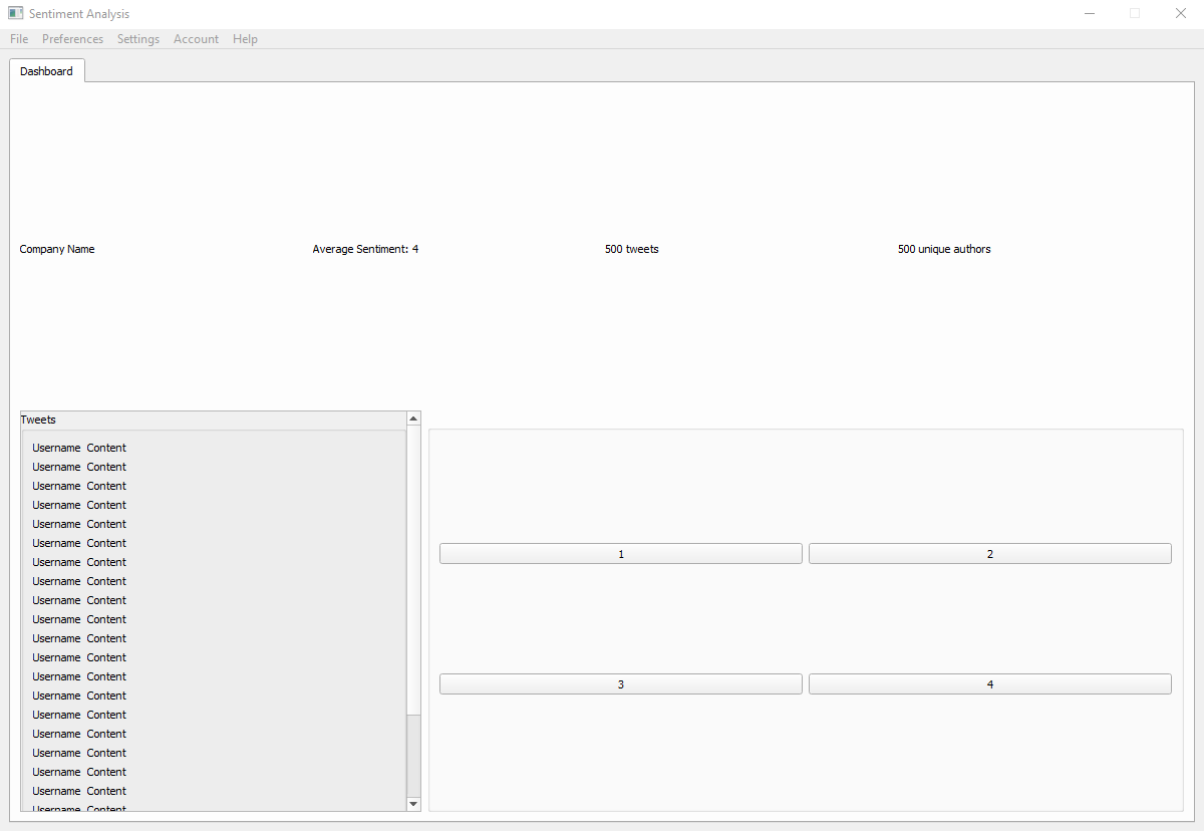


These small tests show that the “add_tab” method within the “TabHandler” class is successful. Now I know this works as intended, I can add the visual elements required for different tab types.



Looking at the design for the main window, all of the elements on screen are dynamic so I will handle what is shown on these windows in the next stage where I am handling integration. To demonstrate what the tabs may look like in future, I have added

```
class Tab(QWidget):
    def __init__(self):
        super(Tab, self).__init__()

    def initUI(self):
        self.windowLayout = QVBoxLayout()

        # TOP LAYOUT:
        self.topLayout = QHBoxLayout()
        self.topLayout.addWidget(QLabel(str(self.name).title()))
        self.topLayout.addWidget(QLabel("Average Sentiment: 4"))
        self.topLayout.addWidget(QLabel("588 tweets"))
        self.topLayout.addWidget(QLabel("588 unique authors"))

        # BOTTOM LAYOUT:
        # scroll box:
        self.bottomLayout = QHBoxLayout()
        formLayout = QFormLayout()
        groupBox = QGroupBox("Tweets")
        usernames = ["Username" for i in range(25)]
        tweets = [QLabel("Content") for i in range(25)]
        for i in range(25):
            formLayout.addRow(usernames[i], tweets[i])
        groupBox.setLayout(formLayout)
        scroll = QScrollArea()
        scroll.setWidget(groupBox)
        scroll.setWidgetResizable(True)
        scroll.setFixedWidth(400)
        scroll.setFixedHeight(400)
        self.bottomLayout.addWidget(scroll)

        # grid layout:
        self.horizontalGroupBox = QGroupBox()
        self.gridLayout = QGridLayout()
        # self.gridLayout.setColumnStretch(0, 4)

        self.gridLayout.addWidget(QPushButton('1'), 0, 0)
        self.gridLayout.addWidget(QPushButton('2'), 0, 1)
        self.gridLayout.addWidget(QPushButton('3'), 1, 0)
        self.gridLayout.addWidget(QPushButton('4'), 1, 1)

        self.horizontalGroupBox.setLayout(self.gridLayout)
        self.bottomLayout.addWidget(self.horizontalGroupBox)

        self.windowLayout.addLayout(self.topLayout)
        self.windowLayout.addLayout(self.bottomLayout)

        self.setLayout(self.windowLayout)


class Dashboard(Tab):
    def __init__(self, company):
        super(Dashboard, self).__init__()
        self.name = "company name"
        print(self.name)
        self.initUI()


class TopicTab(Tab):
    def __init__(self, topic):
        super(TopicTab, self).__init__()
        self.name = topic.name
        self.initUI()
```



This is starting to resemble what the tabs should look like (again, I have made up the data to show what it could look like). In the next stage of development, I will integrate the database and show user data on this window. And improve how the window looks visually.

Stage 4 – Reflection

What has been completed?

In this stage, the primary components of the user interface have been completed. This includes database integration with components like the login and registering system. The main window is also able to show data relevant to the user account used to sign in.

Overview of the prototype as a whole

Currently, when run, the prototype displays the login page with no means of accessing the other windows of the application. This means the main application file is not yet accessing the other components of the system and so it is currently unusable at this stage. The next stage will involve database integration with the user interface including the ability to login and allows the user to access their own data.

What has been tested and how?

Testing during this stage primarily running the code and looking at the displayed user interface then comparing with the designs proposed in the development section. When documenting, this comparison was represented using simple tables showing screenshots of the two designs.

Links to the success criteria

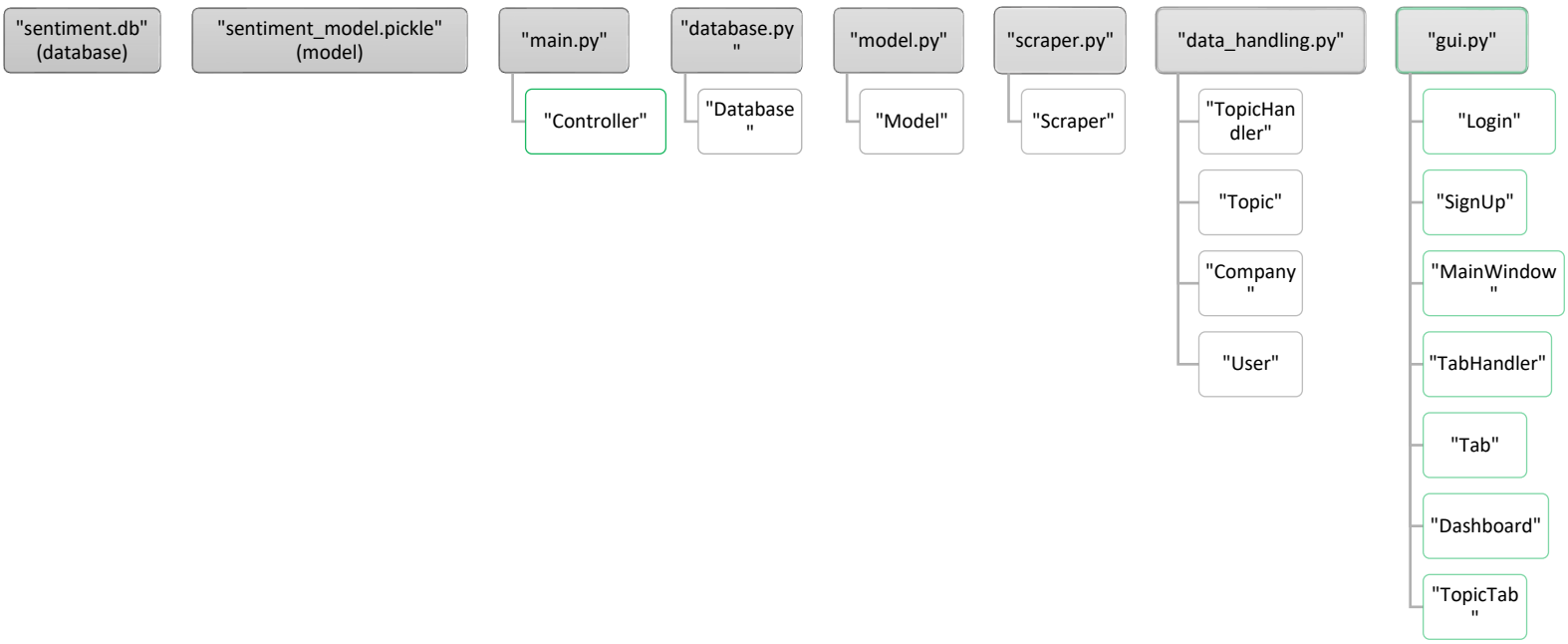
- *"A login/sign up window is provided prompting the user to enter their details"*

- "The software will be displayed on a large main window so that data is easy to read"
- "The main window has a dashboard containing analysis on the user's company"
- "Where possible, data will be shown graphically."
- "There will be a settings menu where the user can adjust the software settings"
- "There will be a help screen where users can find out how to use the software and find out more about the software"

Are there any changes that may need to be made to this component further into development?

Although the visual user interface elements themselves are fully completed, and navigation between the various UI components is now possible, the system still lacks the ability to use the other components like the model and/or scraper to make decisions on what to show on the screen. This means that further development of the user interface will primarily involve integration rather than further building more window designs.

Current system structure (new items have a green outline)



Stage 5 – User Interface system integration

Login and signup integration

Now I have successfully running login and signup forms and a main window, I can update the controller class so that I can jump between each of these windows.

```
class Controller:  # Controls what is shown on the user interface
    def __init__(self):
        self.widget = QtWidgets.QStackedWidget()  # Create the window stack

        self.login = Login(self)  # Instantiates the login screen
        self.widget.addWidget(self.login)  # Adds the login to the stack

        self.signup = SignUp(self)
        self.widget.addWidget(self.signup)

        self.mainWindow = MainWindow(self)  # Instantiates the main window (don't need to add to the stack)

        self.show_login()  # Runs the "show_sign_up" method, showing the sign up form

    def show_main_window(self):  # Is run to show the main window
        self.widget.hide()  # Hide the widget stack
        self.mainWindow.initUI()  # Sets up the user interface elements of the main window

    def show_sign_up(self):  # Is run to show the signing up screen
        self.signup.initUI()  # Sets up the user interface elements of the sign up screen
        self.widget.setCurrentIndex(1)  # Sets the index of the stack so the sign up screen is shown
        self.widget.show()  # Shows the updated stack (i.e shows the login screen)

    def show_login(self):  # Is run to show the login screen
        self.login.initUI()
        self.widget.setCurrentIndex(0)
        self.widget.show()
```

Now I have a working login system I need to make use of the classes created in the first stage that handle the user's data. This will require modification to the Controller class.

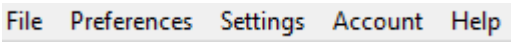
```
class Controller:
    def __init__(self):
        # Create the window stack
        self.widget = QtWidgets.QStackedWidget()

        # Create the login and signup windows and add them to the widget stack
        self.login = gui.Login(self)
        self.login.initUI()
        self.widget.addWidget(self.login)

        self.signup = gui.SignUp(self)
        self.signup.initUI()
        self.widget.addWidget(self.signup)

        # Show the current window
        self.widget.show()

    def show_main_window(self, username):
        # Hide the login and sign up window stack
        self.widget.hide()

        # Begin the main function
        self.main(username)

    def show_sign_up(self):
        # Change the window title
        self.widget.setWindowTitle("Sign Up")

        # Change stack index to show the sign up screen
        self.widget.setCurrentIndex(1)

    def show_login(self):
        # Change the window title
        self.widget.setWindowTitle("Login")

        # Sets the index of the stack so the login screen is shown
        self.widget.setCurrentIndex(0)

    def main(self, username):

        # Instantiate the user class
        self.user = User(username)

        # Sets up the user interface elements of the main window
        self.mainWindow = gui.MainWindow(self, self.user)
        self.mainWindow.initUI()
```

I have added function simply called “main” which will be where most of the duration of the software’s runtime will be spent. Initially, it uses the username which the user had entered when logging in, to instantiate the “User” class that I had created in the first stage and updated . When instantiated, it uses the class’ methods to create a “Company” instance as well as collating the users topics and creating “Topic” instances. Using the lines highlighted in red boxes in the screenshot below, showing the “Topic” and “Company” classes, I immediately perform sentiment analysis on

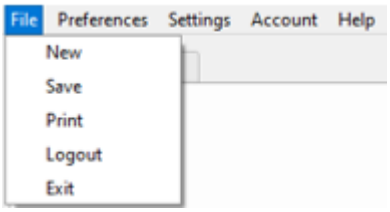
the company/topics and make logs on the results of this data. This means I can immediately use the initial "User" instance to access all of the necessary data to start presenting data on the user interface.

Menu bar integration

Now I need add functionality to the menu bar shown on the main window.



File menu



Looking at the designs, "Topic Manager" needs to provide the user with a window to manage their current topics being analysed. This should include option to add new topics to be analysed and removing those that are no longer needed. Once the user adds/removes a topic, there are three things that need to happen. First, the relationship between the user and the topic in the database needs to be removed so the system can recognise they are no longer related Next, the tabs on the main window need to be updated, adding new topic tabs or removing those that are no longer needed. Lastly, the topic manager window itself needs to update, only showing topics that need to be.

First, I will create a method within the "Database" class that updates the main window tabs. I already have made functions for adding and removing tabs so this method will only need to iterate through the user's topic and ensure all of the current topics and being shown. The following function which is a part of the "TabHandler" class created earlier can handle this.

```
def update_tabs(self):

    # Adding tabs
    for topic in self.parent.user.topics:
        if topic.name.lower() not in [tab.name.lower() for tab in self.tabsList]:
            self.add_new_tab(topic)

    # Deleting tabs
    for tab in self.tabsList:
        if tab.name.lower() not in [topic.name.lower() for topic in self.parent.user.topics]:
            self.remove_tab(tab)
```

The variable in the red box is the instance of the "User" class that I was explaining earlier. This is an example of how the user's attributes can be used in other areas of the software.

Now, I can also create the class which will prompt the user to add and remove new topics to the system. Below is the "Topic Manager" class within "database.py":

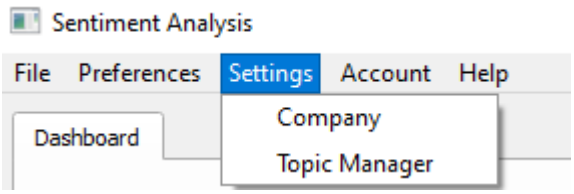
```python
class TopicManagerPopUp(QWidget):
    def __init__(self, parent):
        super().__init__()
        self.parent = parent
        self.initUI()

    def initUI(self):
        # Set title of popup
        self.setWindowTitle("Topic Manager")

        # Set dimensions of popup
        width = 400
        height = int(width * 9 / 16)
        self.setGeometry((self.parent.width()/2 - width/2), (self.parent.height()/2 - height/2), width, height)
        self.setFixedSize(width, height)

        # Create the layout for the popup
        self.popUpLayout = QGridLayout()
        self.setLayout(self.popUpLayout)

        # Add the layout for creating a new topic
        newTopicLayout = QVBoxLayout()
        newTopicLayout.setAlignment(Qt.AlignVCenter)

        newGroupBox = QGroupBox("Create New Topics")
        newGroupBox.setAlignment(Qt.AlignHCenter)

        self.newTopicField = QLineEdit()
        newTopicLayout.addWidget(self.newTopicField)

        newTopicButton = QPushButton("Add new topic")
        newTopicButton.clicked.connect(self.new_topic)
        newTopicButton.setShortcut('Return')
        newTopicLayout.addWidget(newTopicButton)


        newGroupBox.setLayout(newTopicLayout)

        newScrollBox = QScrollArea()
        newScrollBox.setWidget(newGroupBox)
        newScrollBox.setWidgetResizable(True)

        self.popUpLayout.addWidget(newScrollBox, 0, 1)

        # Add the layout showing the current topics
        self.update_topics_shown()


    def remove_topic(self, button_id):
        # Iterate through the buttons
        for index, button in enumerate(self.buttonList):
            if button_id == button:
                # Remove the topic on the row the button was clicked
                if index < len(self.parent.user.topics):
                    self.parent.user.remove_topic(self.parent.user.topics[index])

        # Update the main window tabs
        self.parent.tab_widget.update_tabs()
        self.update_topics_shown()

    def new_topic(self):
        # Get the name of the new entered topic
        topicName = self.newTopicField.text()
        self.newTopicField.setText("")

        # Check if topic already exists
        if topicName.lower() not in [str(topic.name).lower() for topic in self.parent.user.topics] and topicName != "":
            # Create the new topic
            self.parent.user.new_topic(topicName.lower())

            # Update the main window tabs
            self.parent.tab_widget.update_tabs()
        self.update_topics_shown()

    def update_topics_shown(self):
        # Get the current user topics
        topics = self.parent.user.topics

        topicFormLayout = QFormLayout()
        self.buttonList = []

        # Add each of the topics with a remove button next to them
        for index, topic in enumerate(topics):
            newTopicButton = QPushButton("Remove topic")
            newTopicButton.clicked.connect(lambda: self.remove_topic(self.sender()))
            self.buttonList.append(newTopicButton)
            topicFormLayout.insertRow(index, QLabel(f"{topic.name}"), newTopicButton)

        # Add the box showing the topics
        self.removalGroupBox = QGroupBox("Existing Topics")
        self.removalGroupBox.setAlignment(Qt.AlignHCenter)

        self.removalScroll = QScrollArea()
        self.removalScroll.setWidgetResizable(True)

        # Add the layout for viewing and removing existing topics
        self.removalGroupBox.setLayout(topicFormLayout)

        # Add the scroll box to the window
        self.removalScroll.setWidget(self.removalGroupBox)

        # Add to the window
        self.popUpLayout.addWidget(self.removalScroll, 0, 0)
```

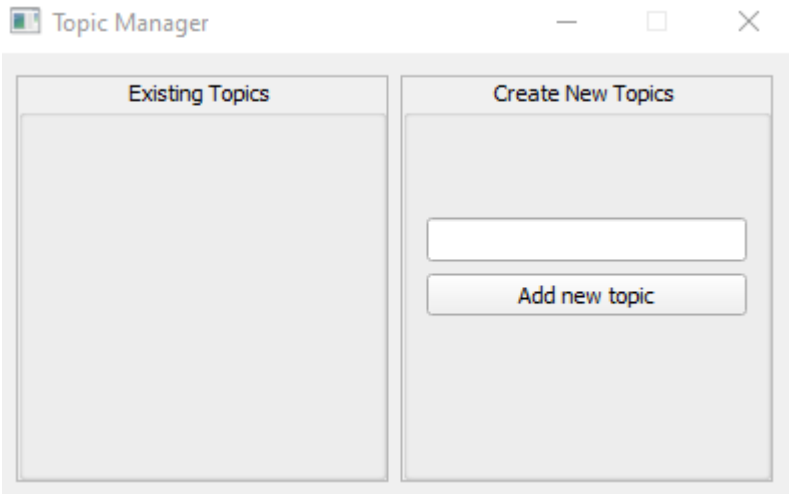
The class mostly makes use of existing methods in other classes such as “update_tabs” from the “Tab_Handler” class that was written earlier as well as the “remove_topic” and “new_topic” which are both methods from the “User” class, allowing me to directly change the user’s topics from this topic manager.

```
class MainWindow(QMainWindow):
    def __init__(self, parent, user):...

    def initUI(self):  # Initialise the UI
        self.setWindowTitle('Sentiment Analysis')

        # Establish geometry
        width = 1500
        height = int(width * 9 / 16)

        # Centre the window
        self.setGeometry(get_screen_size()[0] / 2 - width / 2, get_screen_size()[1] / 2 - height / 2, width, height)

        # Maximise the window
        self.showMaximized()

        # Add menu
        self.add_menu()

        # Add Tabs
        self.tab_widget = TabHandler(self)
        self.setCentralWidget(self.tab_widget)

        # Create a tab manager instance
        self.popUp = TopicManagerPopUp(self)
        self.popUp.hide()

        self.palette = self.palette()
        self.setPalette(self.palette)

        self.show()  # Shows the main window on the screen

    def add_menu(self):...

    def manage_topics(self):
        # Show tab manager popup if not already being shown
        if self.popUp.isHidden():
            self.popUp.show()

    def change_company(self):...

    def user_guide(self):...

    def about(self):...

    def sign_out(self):...

    def change_theme(self):...

    def save(self):...
```

This class is instantiated in the “MainWindow” class create earlier:

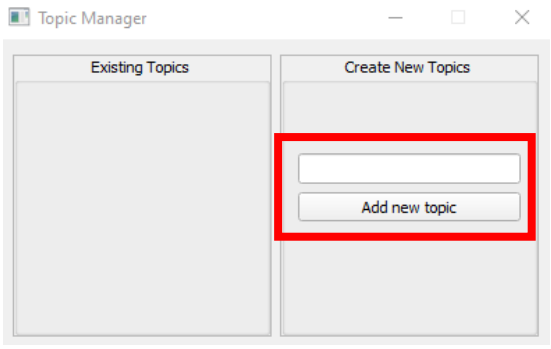
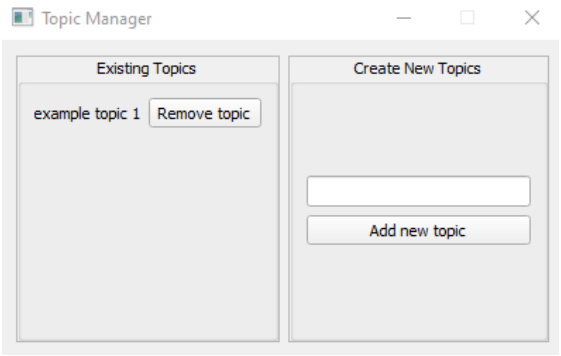
Now when, “Topic Manager” option is clicked under the “Settings Menu” the window shown in the second screenshot is displayed (without the main window closing):



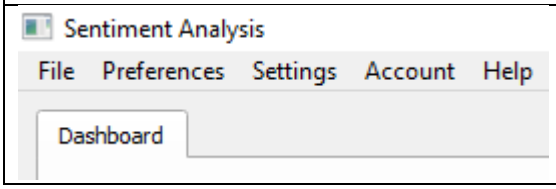
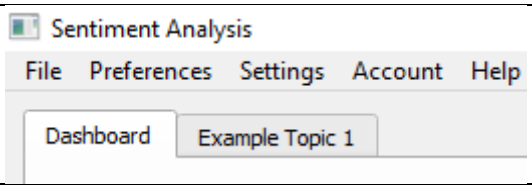


To test how adding topics affects the main window and topic manager window, I typed “Example topic 1” and then clicked the “Add new topic” button. These are the results:

(Topic manager window)

Before	After
	

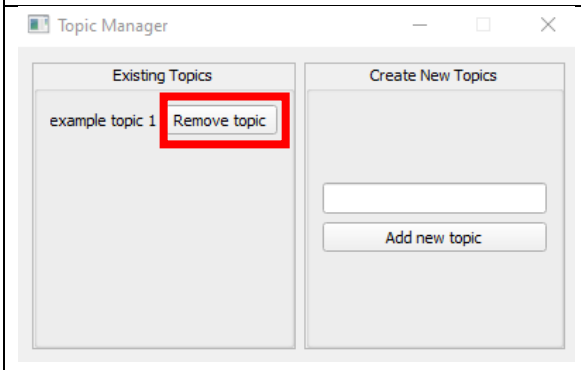
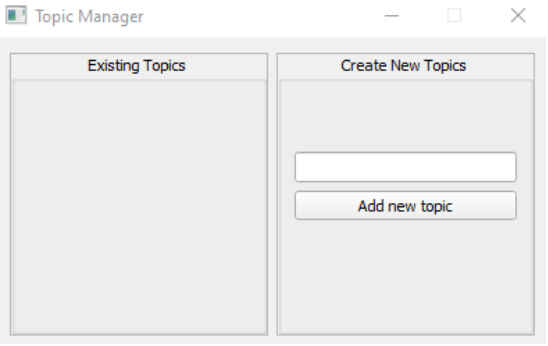
(Main window)

Before	After
	

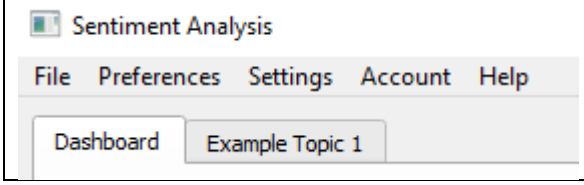
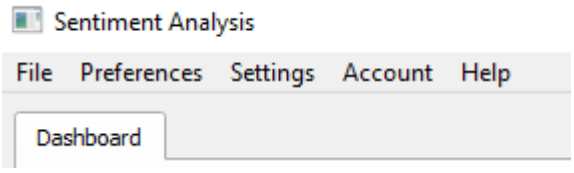
This shows that the user interface is capable of visually updating the user topics.

To test how removing topics affects the main window and topic manager window, I clicked “Remove topic” on the “example topic 1” row. These are the results:

(Topic manager window)

Before	After
	

(Main window)

Before	After
	

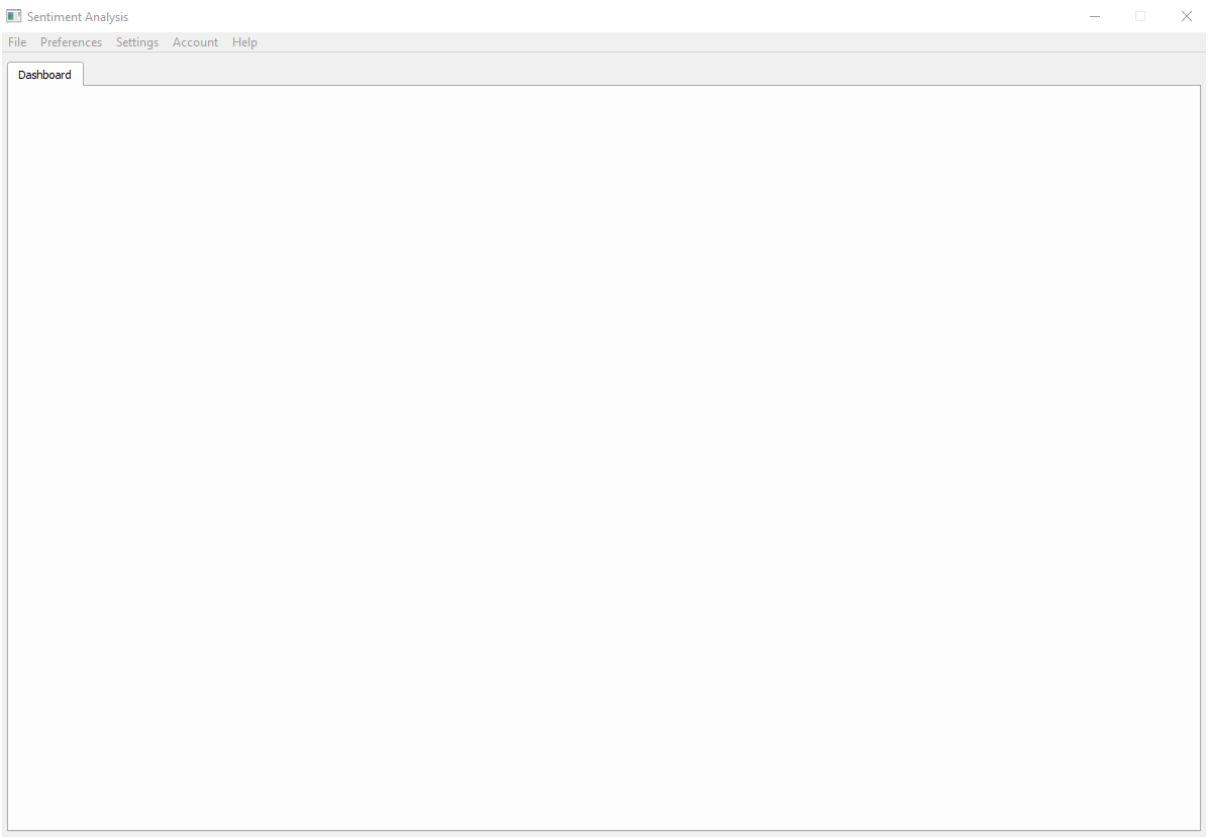
This shows that the system is also capable of updating the UI when the user chooses to remove topics. This means the tab system is almost complete, now I just need to write the content of the tabs which will be done once the menu is complete.

I also need the user to be able to logout if necessary. The following function allows for this, resetting the current user of the system and also removing the text on the login and register windows that the user entered when they first logged in (located within the “MainWindow” class.

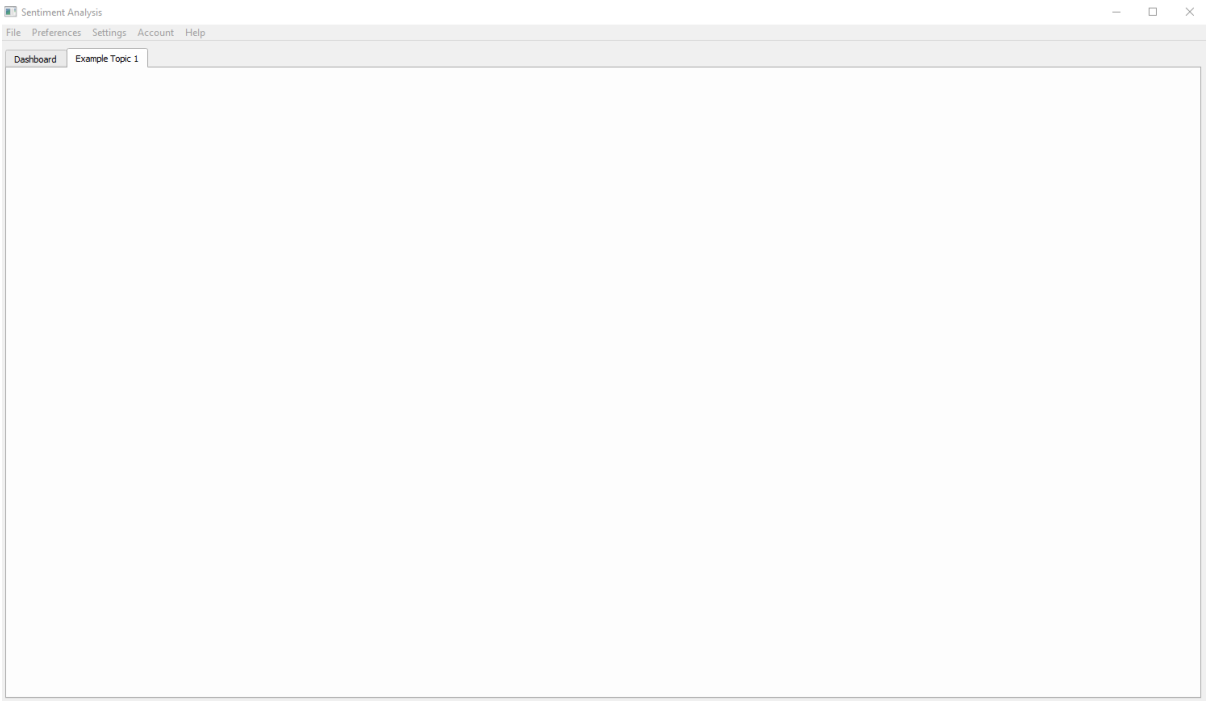
```
def sign_out(self):
    self.close()

    time.sleep(0.25)

    self.parent.widget.show()
    self.parent.username = None

    self.parent.login.username.setText("")
    self.parent.login.password.setText("")
    self.parent.signup.username.setText("")
    self.parent.signup.password.setText("")
    self.parent.signup.company.setText("")
```

Tab content

Currently the main window dashboard looks like this:



And any topic tabs I add using the topic manager look like this:



Now I will write the content of each tab. (As a preface, all of the data being shown for this stage will be shown from my admin account for the system which has the username “Admin” and the company name “Admin” so the data shown here isn’t entirely representative of what could be shown for a user with an actual appropriate company name.)

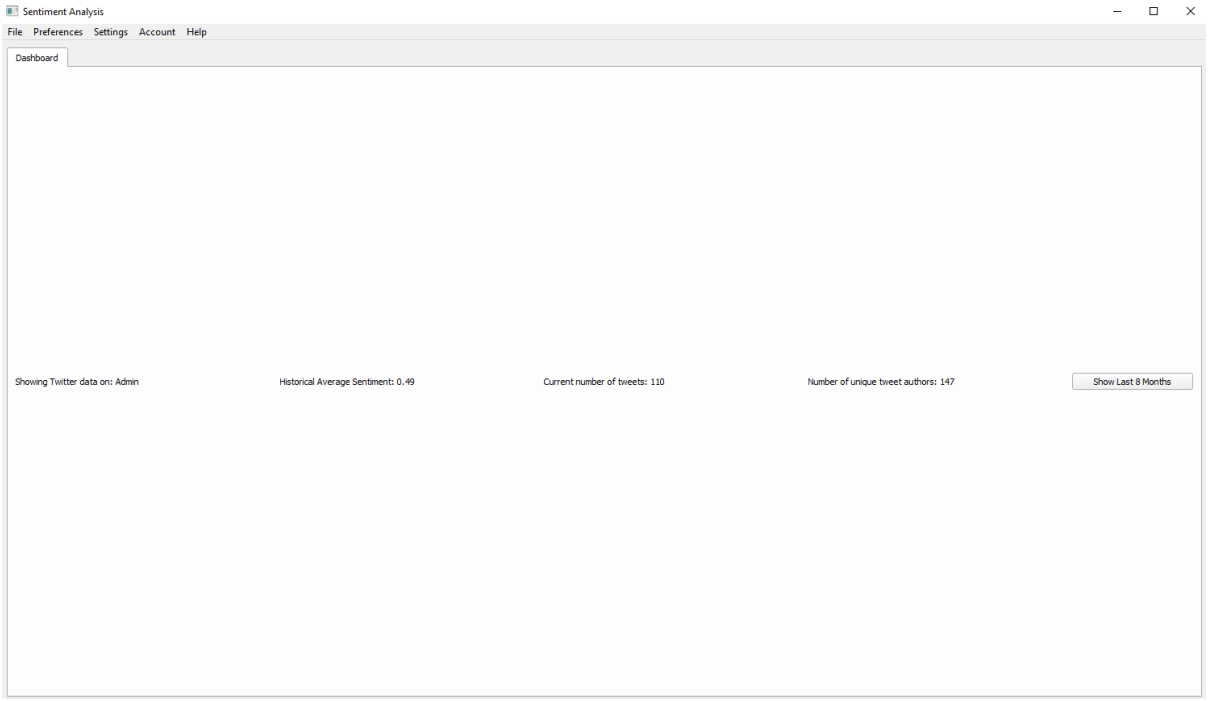
The tab needs to access the data that belons to each topic it is representing and the database so it will initially take the topic or company to be represented as a parameter.

```
class Dashboard(Tab):
    def __init__(self, company):
        super().__init__(company)



class TopicTab(Tab):
    def __init__(self, topic):
        super().__init__(topic)
```

The tab then initialises the UI components using the parent class method shown below.

```
class Tab(QWidget):
    def __init__(self, topic):
        super(Tab, self).__init__()

        # Get all of the details of the topic to be shown
        self.name = topic.name

        # Establish the time scale of the graphs being shown (last 7 days or last 8 months)
        self.timeScale = "week"

        # Data from the analyses that took place upon logging in
        self.tweets = topic.tweets
        self.likes = topic.likes
        self.authors = topic.authors
        self.predictions = topic.predictions
        self.sentiments = topic.sentiments

        # Get the topic data from last 7 days/last 8 months
        self.historicalAverageSentiment = topic.historicalAverageSentiment
        self.monthsTweets = topic.monthsTweets[::-1]
        self.monthsAverageSentiments = topic.monthsAverageSentiments[::-1]
        self.monthsAveragePosNegTweets = topic.monthsAveragePosNegTweets[::-1]
        self.lastWeeksTweets = topic.lastWeeksTweets[::-1]
        self.lastWeeksAverageSentiments = topic.lastWeeksAverageSentiments[::-1]
        self.lastWeeksAveragePosNegTweets = topic.lastWeeksAveragePosNegTweets[::-1]

        # Initialise the UI components of the tab
        self.initUI()

    def initUI(self):
        self.windowLayout = QVBoxLayout()

        # Add the top layout
        self.topLayout = QHBoxLayout()
        self.topLayout.addWidget(f"Showing twitter data on: {QLabel(str(self.name))}")
        self.topLayout.addWidget(QLabel(f"Historical Average Sentiment: {str(self.historicalAverageSentiment)}"))
        if len(predictions) != 0:
            self.topLayout.addWidget(QLabel(f"Current number of tweets: {str(len(self.tweets))}"))
            self.topLayout.addWidget(QLabel(f"Number of unique tweet authors: {str(len(set(self.authors)))}"))
        self.scaleButton = QPushButton("Show Last 8 Months")
        self.scaleButton.setFixedWidth(150)
        self.scaleButton.clicked.connect(self.change_time_scale)
        self.topLayout.addWidget(self.scaleButton)
        self.windowLayout.addLayout(self.topLayout)

        # Create the bottom layout
        self.bottomLayout = QHBoxLayout()
        self.bottomLayout.setAlignment(Qt.AlignLeft)

        # Add the charts/graphs to the tab
        self.graphLayout = QGridLayout()

        # Graphs go here

        # Combine layouts
        self.bottomLayout.addLayout(self.graphLayout)
        self.windowLayout.addLayout(self.bottomLayout)

        # Add the layouts to the window
        self.setLayout(self.windowLayout)

    def change_time_scale(self):
        # Change the time scale of the graphs being shown
        if self.timeScale == "year":
            self.timeScale = "lastWeek"
            self.scaleButton.setText("Show Last 8 Months")
        else:
            self.timeScale = "year"
            self.scaleButton.setText("Show Last 7 Days")

        # Plot the updated graphs
        self.plot_tweets_over_time()
        self.plot_pos_neg_over_time()
        self.plot_sent_over_time()
```

So the main window now looks like this with dynamic labels:



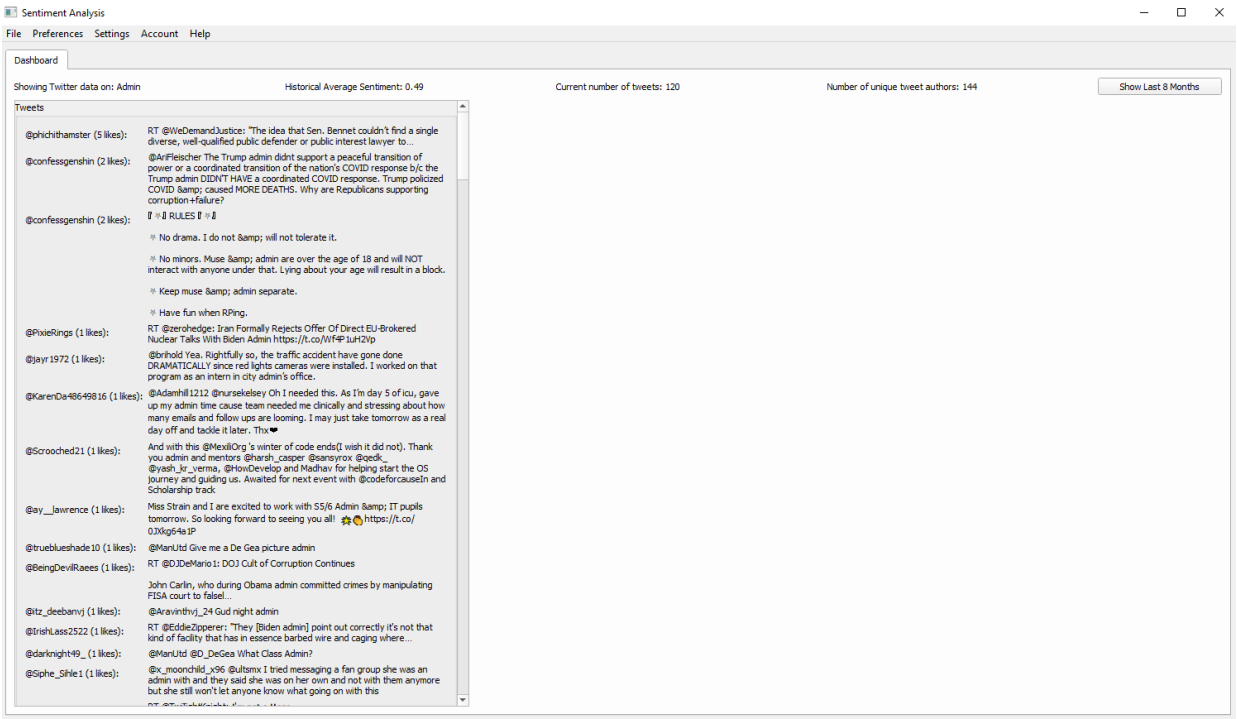
The “Show Last 8 Months” button and “change_time_scale” function is to allow for switching between showing data from the last 7 months and the last 7 days.

I will now add some functions to the class that will be called in this “initUI”. The first will be a box containing tweets about the company/topic on that tab.

The following function achieves this:

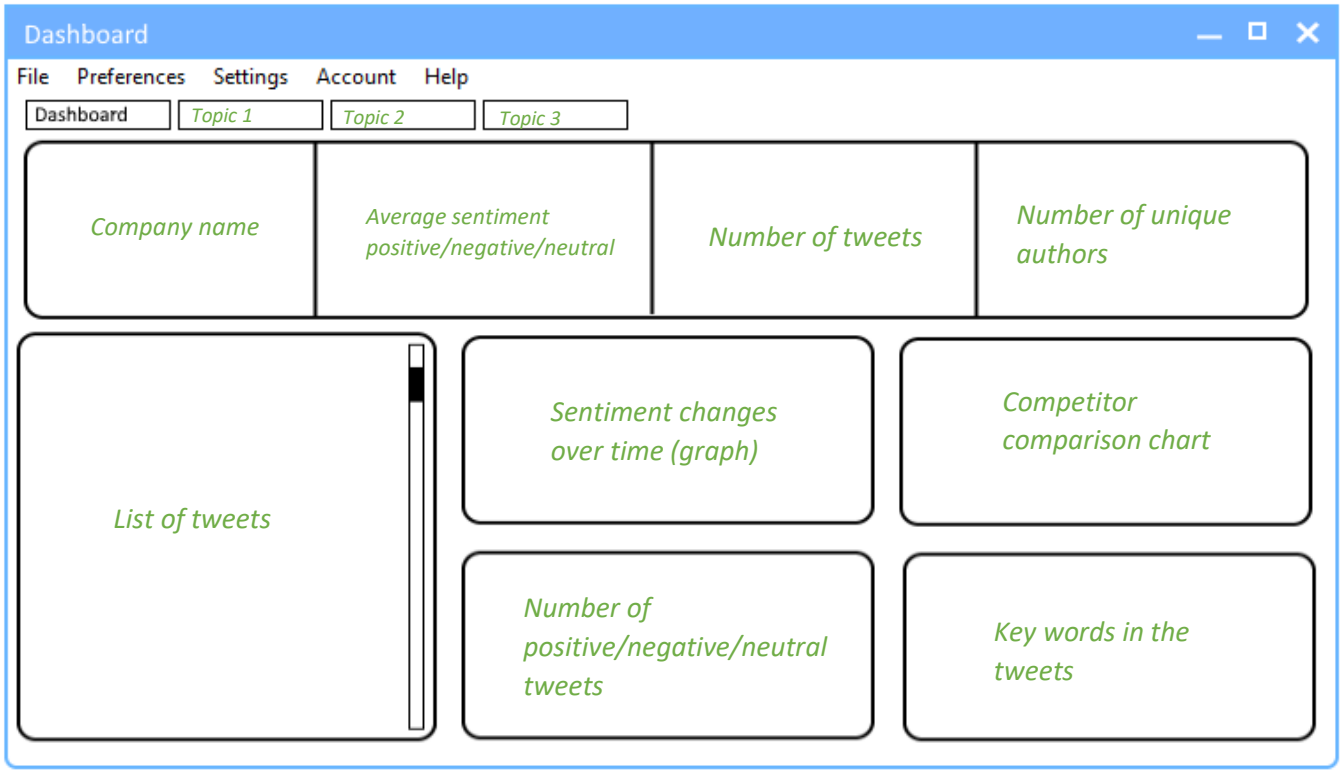
```
def add_scroll_box(self):

    # Create a scroll box containing tweets about the topic/company
    formLayout = QFormLayout()
    groupBox = QGroupBox("Tweets")
    groupBox.setMaximumWidth(550)

    # Order the tweets according to how many likes they have
    zippedTweets = zip(self.tweets, self.likes, self.authors)
    zippedTweetsList = list(zippedTweets)
    sortedTweets = sorted(zippedTweetsList, key=lambda x: x[1], reverse=True)

    # Add the ordered tweets to the layout
    for i in range(len(sortedTweets)):
        formLayout.addRow(QLabel(f"@{sortedTweets[i][2]} ({sortedTweets[i][1]} likes):"),
                          QLabel(sortedTweets[i][0], wordWrap=True, alignment=Qt.AlignLeft | Qt.AlignTop))
    groupBox.setLayout(formLayout)

    # Create a scroll box to put the tweets in
    scroll = QScrollArea()
    scroll.setWidgetResizable(True)
    scroll.setMaximumWidth(550)
    scroll.setWidget(groupBox)

    # Add this scroll box to the window
    self.bottomLayout.addWidget(scroll)
```

Calling this function in the “initUI” results in the main window looking like this:



(As said above, the tweets shown are not based on a particular company, but my Admin profile so these tweets are talking about the word “Admin”. Going forward, I will be hiding the tweets in the screenshots I provide within this document to avoid accidentally showing anything offensive)

Adding graphs



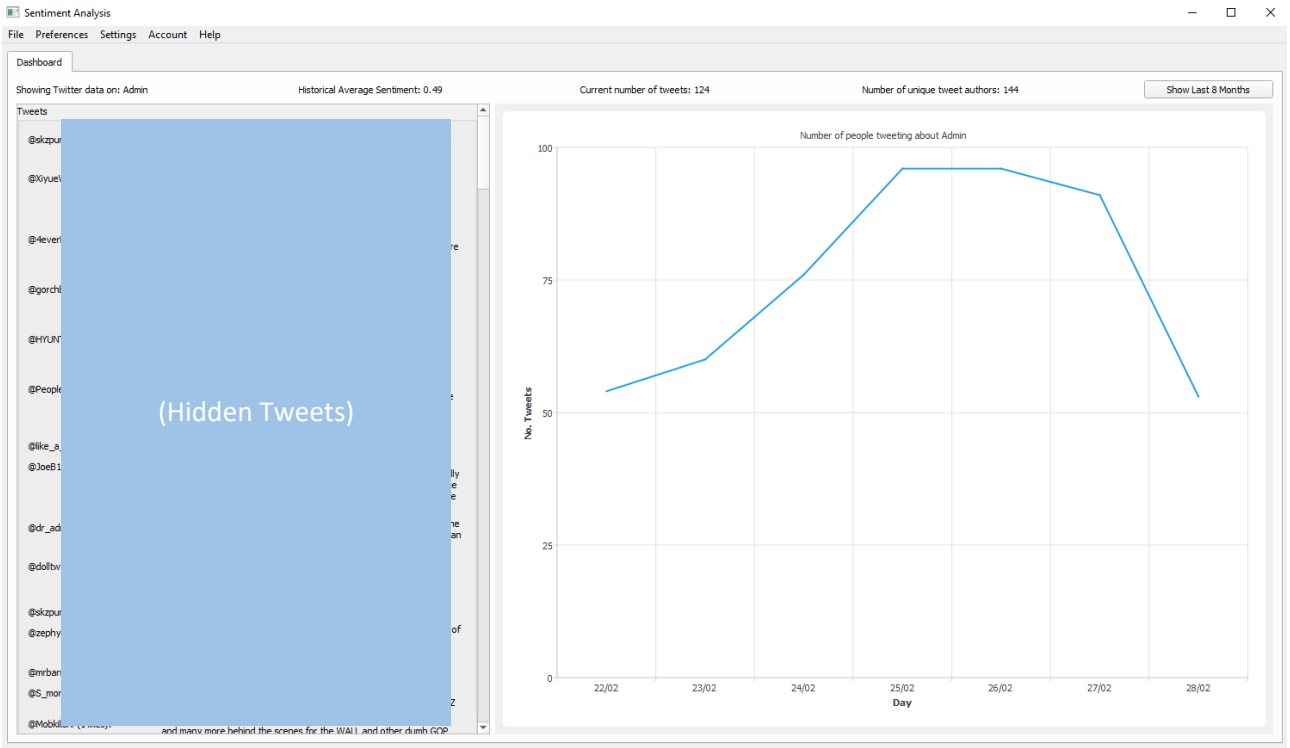
Now I will add the necessary graphs to the main window tabs. Looking at the success criteria and initial design, I need graphs to show changes in data over time. For now, I will be adding graphs showing changes over the last 7 days and the last 8 months.

First, I will add the graphs showing the time series data i.e changes over the last 7 days and 8 months. The first graph I will add is a line graph showing the average number of tweets about the user's company in a given time period.

```
def plot_tweets_over_time(self):

    # Create a line graph showing the average number of tweets per month
    series = QLineSeries()

    # Create the data to go on the graph
    if self.timeScale == "year":
        data = [QPoint(month, noTweets) for month, noTweets in enumerate(self.monthsTweets)]
        xLabels = [(datetime.now() - timedelta(days=30 * i)).strftime("%m/%y") for i in range(0, 8)][::-1]
        timeScale = "Month"
    else:
        data = [QPoint(day, noTweets) for day, noTweets in enumerate(self.lastWeeksTweets)]
        xLabels = [(datetime.now() - timedelta(days=i)).strftime("%d/%m") for i in range(0, 7)][::-1]
        timeScale = "Day"
    series.append(data)

    # Create the graph
    chart = QChart()
    chart.addSeries(series)
    chart.setTitle(f'Number of people tweeting about {self.name}')
    chart.setAnimationOptions(QChart.SeriesAnimations)

    # Add the x-axis
    axisX = QBarCategoryAxis()
    axisX.append(xLabels)
    axisX.setTitleText(timeScale)
    chart.addAxis(axisX, Qt.AlignBottom)
    series.attachAxis(axisX)

    # Add the y-axis
    axisY = QValueAxis()
    axisY.setLabelFormat("%i")
    axisY.setTitleText("No. Tweets")
    chart.addAxis(axisY, Qt.AlignLeft)
    series.attachAxis(axisY)
    chart.axisY(series).setRange(0, (int(math.ceil(max(self.monthsTweets) / 100)) * 100))

    # Hide the legend
    chart.legend().hide()

    # Add the chart to the window
    chartView = QChartView(chart)
    chartView.setRenderHint(QPainter.Antialiasing)
    chartView.setMinimumSize(100, 100)
    self.timeSeriesLayout.addWidget(chartView, 0, 2)
```

The method checks the current time scale. If the time scale is "weeks", the graph will be made using daily data, if the time scale is "year" the graph will be made using monthly data. After it establishes the time scale of the x-axis, it will create the graph. This will be the same structure to the other time series graphs so I won't go into explaining them individually.

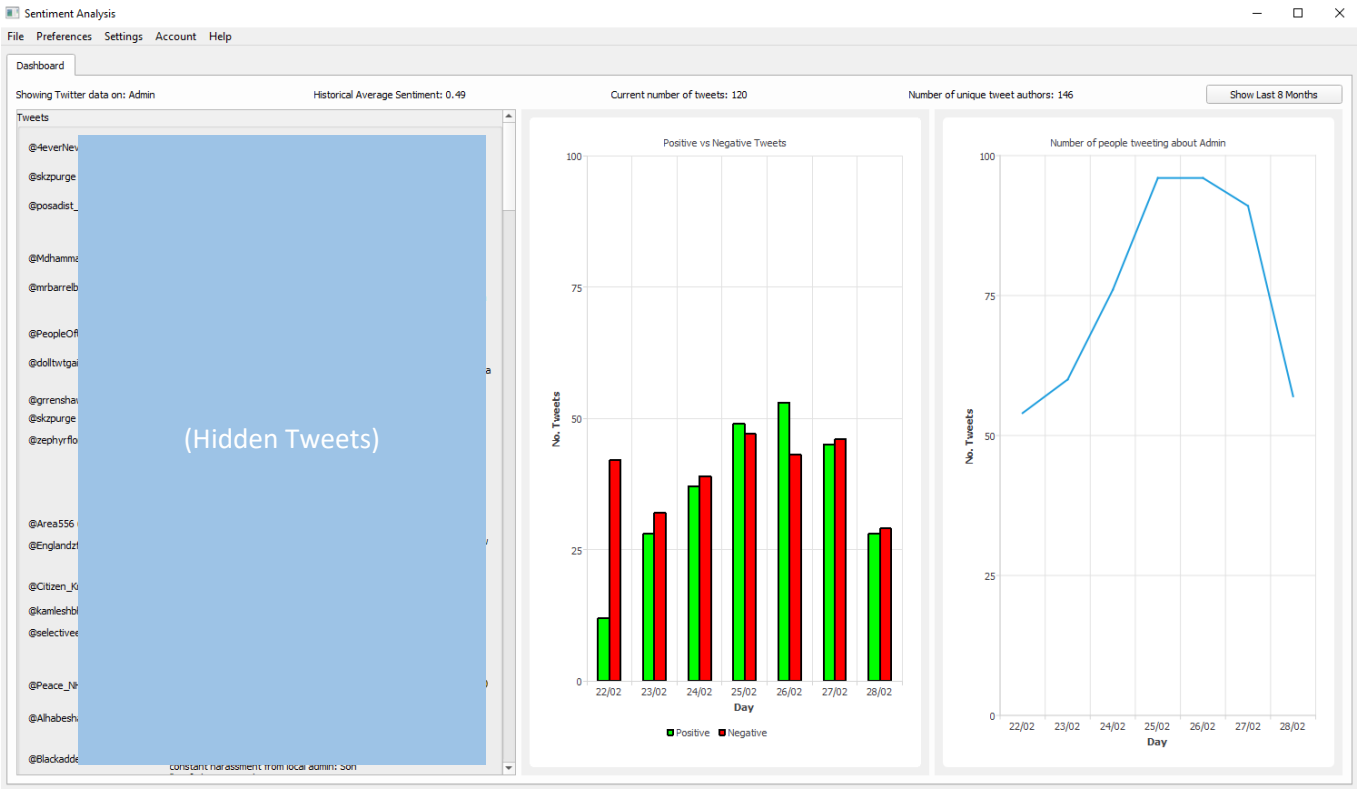
Calling this method in the “initUI” method results in the window looking like this:



Next, I will add a bar chart showing the number of positive and negative tweets over the same time span.

```
def plot_pos_neg_over_time(self):

    # Create bars to go on the bar chart
    posBar = QBarSet("Positive")
    posBar.setColor(Qt.green)
    posBar.setPen(QPen(Qt.black, 2))
    negBar = QBarSet("Negative")
    negBar.setColor(Qt.red)
    negBar.setPen(QPen(Qt.black, 2))

    # Add the data
    if self.timeScale == "year":
        for month in self.monthsAveragePosNegTweets:
            posBar << month[0]
            negBar << month[1]
        xLabels = [(datetime.now() - timedelta(days=30 * i)).strftime("%m/%y") for i in range(0, 8)][::-1]
        timeScale = "Month"
    else:
        for day in self.lastWeeksAveragePosNegTweets:
            posBar << day[0]
            negBar << day[1]
        xLabels = [(datetime.now() - timedelta(days=i)).strftime("%d/%m") for i in range(0, 7)][::-1]
        timeScale = "Day"

    # Add the data
    series = QBarSeries()
    series.append(posBar)
    series.append(negBar)

    # Create the bar chart
    chart = QChart()
    chart.addSeries(series)
    chart.setTitle("Positive vs Negative Tweets")
    chart.setAnimationOptions(QChart.SeriesAnimations)
    chart.setTheme(QChart.ChartThemeLight)

    # Add the y-axis
    axisY = QValueAxis()
    axisY.setLabelFormat("%i")
    axisY.setTitleText("No. Tweets")
    chart.addAxis(axisY, Qt.AlignLeft)
    series.attachAxis(axisY)
    chart.axisY(series).setRange(0, (
                int(math.ceil(max([max(item) for item in self.monthsAveragePosNegTweets]) / 100)) * 100))

    # Add the x-axis
    axisX = QBarCategoryAxis()
    axisX.append(xLabels)
    axisX.setTitleText(timeScale)
    chart.setAxisX(axisX, series)

    # Show the legend
    chart.legend().setVisible(True)
    chart.legend().setAlignment(Qt.AlignBottom)

    # Add the chart to the window
    chartView = QChartView(chart)
    chartView.setMinimumSize(100,100)
    self.timeSeriesLayout.addWidget(chartView, 0, 0)
```

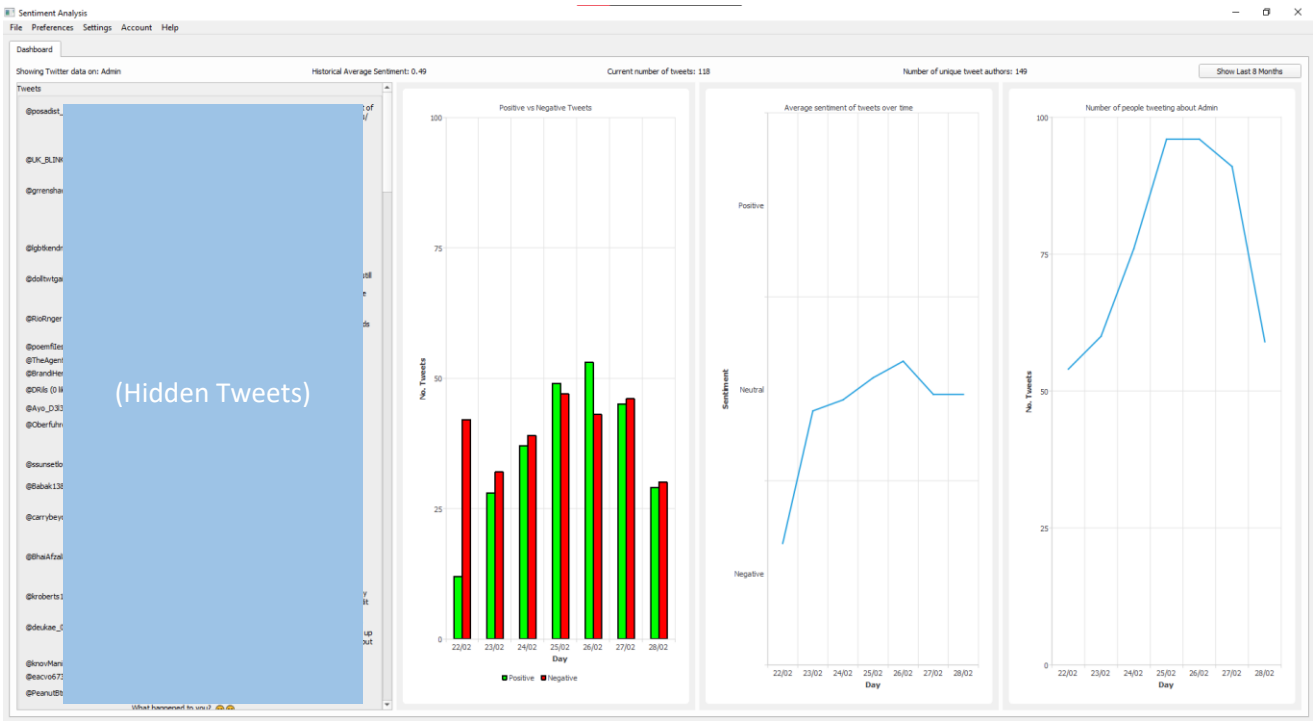
Resulting in:



Next I will add a graph to show the change in average sentiment over time.

```
def plot_sent_over_time(self):

    # Create a line graph showing the average number of tweets per month
    series = QLineSeries()

    # Create the data
    if self.timeScale == "year":
        data = [QPoint(month, int(sentiment * 100)) for month, sentiment in enumerate(self.monthsAverageSentiments)]
        timeScale = "Month"
        xLabels = [(datetime.now() - timedelta(days=30 * i)).strftime("%m/%y") for i in range(0, 8)][::-1]
    else:
        data = [QPoint(day, int(sentiment * 100)) for day, sentiment in enumerate(self.lastWeeksAverageSentiments)]
        timeScale = "Day"
        xLabels = [(datetime.now() - timedelta(days=i)).strftime("%d/%m") for i in range(0, 7)][::-1]
    series.append(data)

    # Create the graph
    chart = QChart()
    chart.addSeries(series)
    chart.setTitle(f'Average sentiment of tweets over time')
    chart.setAnimationOptions(QChart.SeriesAnimations)

    # Add the x-axis
    axisX = QBarCategoryAxis()
    axisX.append(xLabels)
    axisX.setTitleText(timeScale)
    chart.addAxis(axisX, Qt.AlignBottom)
    series.attachAxis(axisX)

    # Add the quantitative y-axis
    axisY = QValueAxis()
    axisY.setLabelFormat("%i")
    axisY.setTickCount(3)
    chart.addAxis(axisY, Qt.AlignLeft)
    series.attachAxis(axisY)
    chart.axisY(series).setRange(0, (int(math.ceil(max(self.monthsAverageSentiments) / 100)) * 100))
    axisY.hide()

    # Replace the y-axis with qualitative labels
    yLabels = ["Negative", "Neutral", "Positive"]
    axisY = QBarCategoryAxis()
    axisY.append(yLabels)
    axisY.setTitleText("Sentiment")
    chart.addAxis(axisY, Qt.AlignLeft)
    axisY.show()

    # Hide the legend
    chart.legend().hide()

    # Add the graph to the window
    chartView = QChartView(chart)
    chartView.setRenderHint(QPainter.Antialiasing)
    chartView.setMinimumSize(100,100)
    self.timeSeriesLayout.addWidget(chartView, 0 , 1)
```

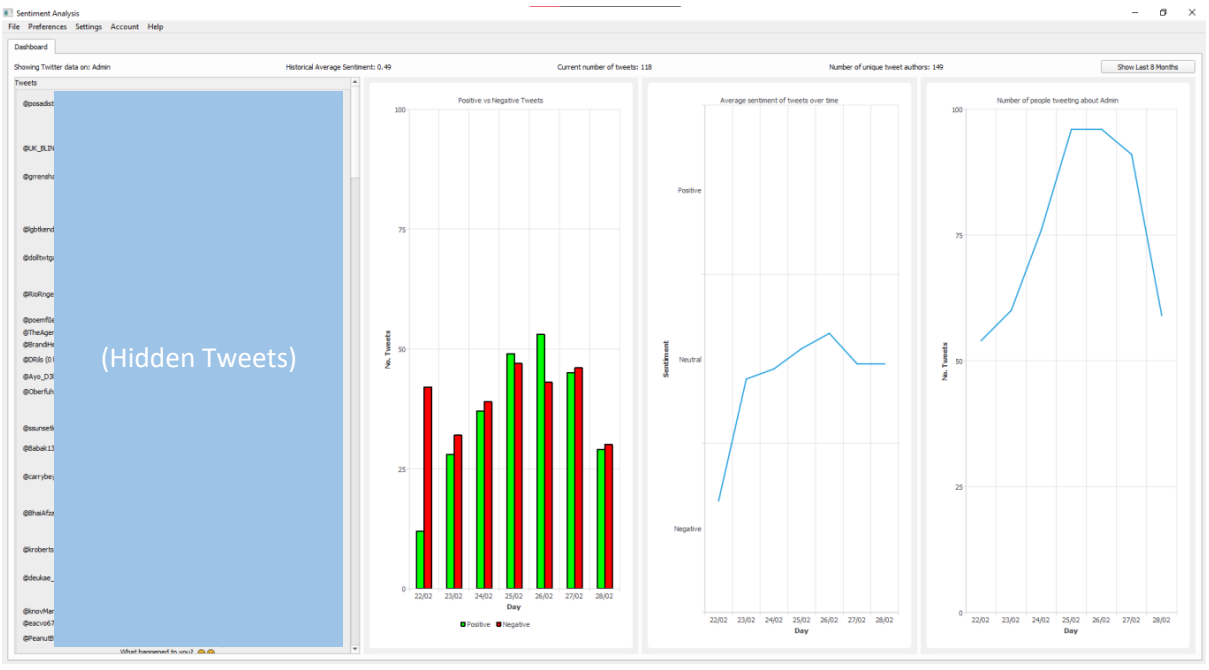
Resulting in:



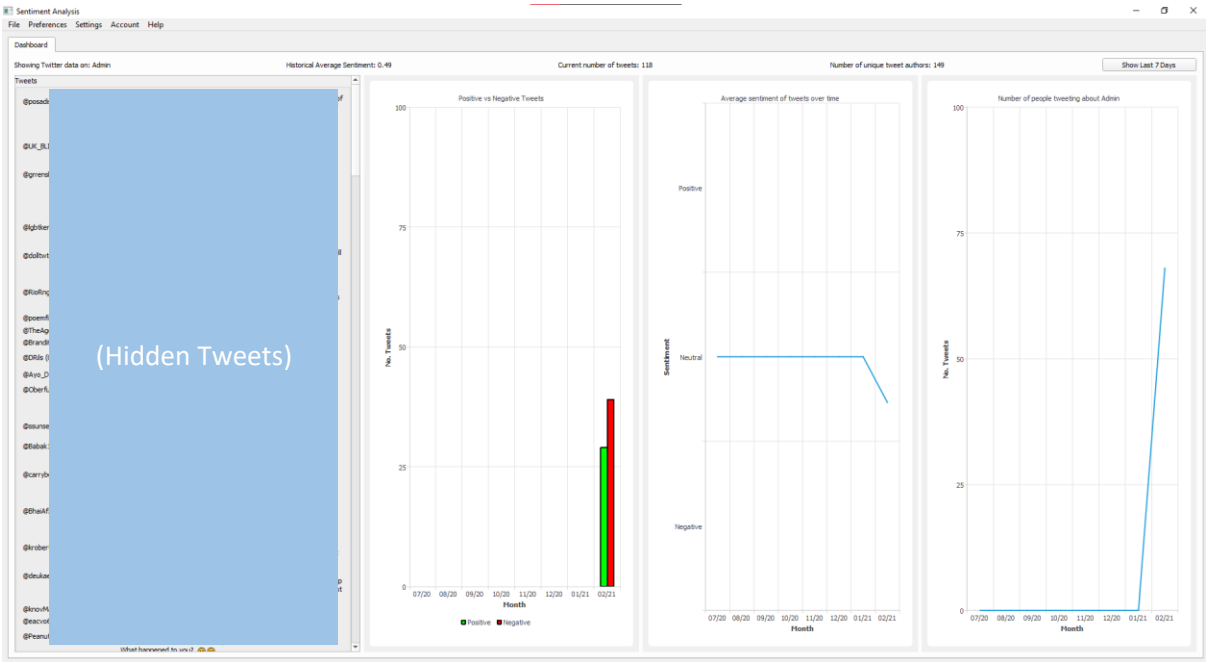
As you can see the text is quite small. I will change the formatting so it is more readable once all of the graphs are put on the window.

Currently the three time series graphs are showing data fluctuations over the last 7 days but if the button indicated by the red box is clicked, the graphs dynamically switch to showing the data from the last 8 months, as shown below.

Before clicking “Show Last 8 Months”:



After clicking “Show Last 8 Months”:



As you can see, the x-axis of the graphs have changed from days to months and the button in the red box that was previously labelled “Show Last 8 Months” is now labelled “Show Last 7 days” which the user can click if they want to switch back to that time period. This shows that changing time period functionality of the system works.

Now I will add some other charts beneath these graphs. First, I will add a pie chart comparing the number of positive and negative tweets in the most recent analysis.

```
def plot_pos_vs_neg(self):

    # Add a pie chart comparing positive and negative tweets from the most recent analysis
    series = QPieSeries()

    # Get the data to put on the pie chart
    pos_percent = neg_percent = 0
    if len(self.tweets) != 0:
        pos_percent = round(self.sentiments.count("Positive") / len(self.sentiments), 2) * 100
        neg_percent = 100 - pos_percent
    series.append("Positive", pos_percent)
    series.append("Negative", neg_percent)

    # Add the positive pie slice
    slice = QPieSlice()
    slice = series.slices()[0]
    slice.setPen(QPen(Qt.black, 2))
    slice.setBrush(Qt.green)

    # Add the negative pie slice
    slice = QPieSlice()
    slice = series.slices()[1]
    slice.setPen(QPen(Qt.black, 2))
    slice.setBrush(Qt.red)

    # Create the pie chart
    chart = QChart()
    chart.legend()
    chart.addSeries(series)
    chart.createDefaultAxes()
    chart.setAnimationOptions(QChart.SeriesAnimations)
    chart.setTitle("Positive vs Negative Tweets")

    # Add the legend to the pie chart
    chart.legend().setVisible(True)
    chart.legend().setAlignment(Qt.AlignBottom)

    # Add the chart to the window
    chartView = QChartView(chart)
    chartView.setRenderHint(QPainter.Antialiasing)
    chartView.setMinimumSize(100, 100)
    chartView.setMaximumWidth(500)
    self.currentDataLayout.addWidget(chartView, 1, 0)
```

After calling this method in the “initUI”:

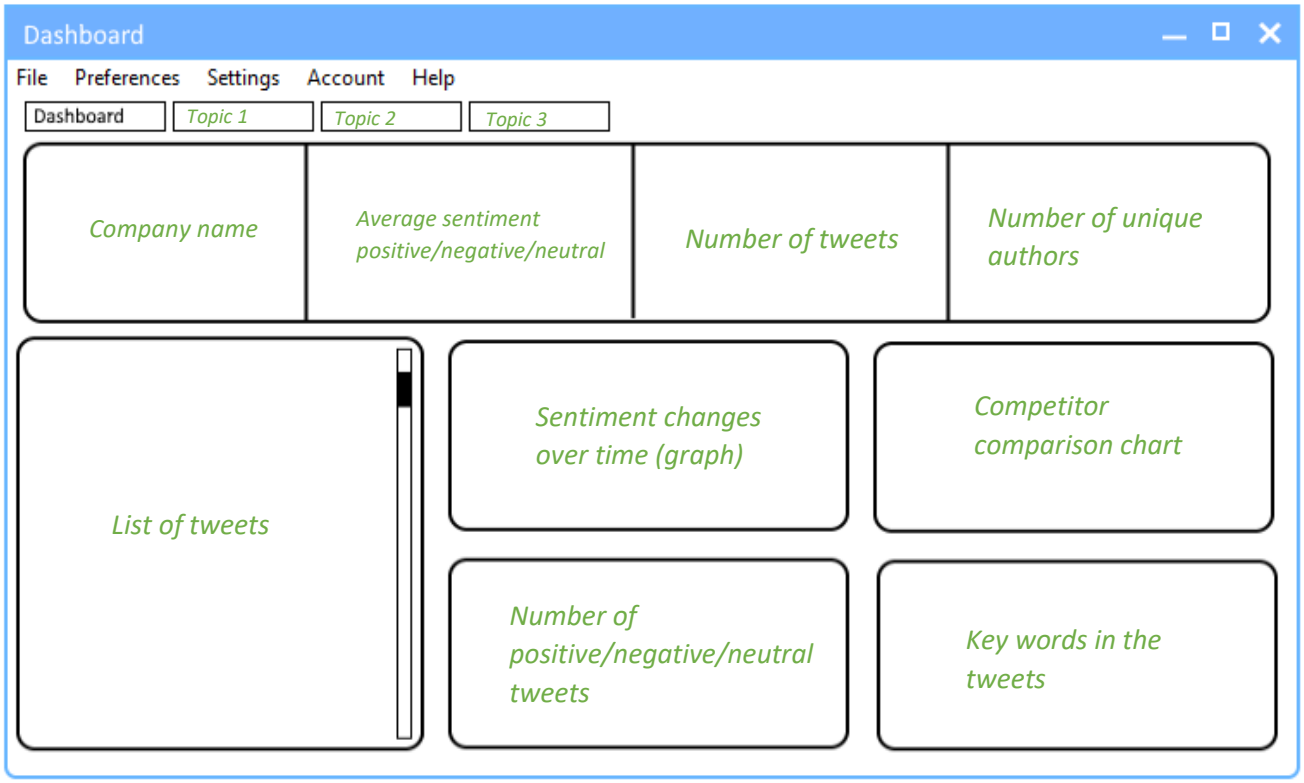
Looking at the success criteria, I also need to add a widget that shows the user the most common words used in tweets about the company/topic. To implement this, I have written the following function, making use of a word cloud to show the data.

```
def plot_word_cloud(self):
    # Get the tweets ready for the word cloud
    tweets = [" ".join([word.strip() for word in tweet.split() if len(word)>2 and "https" not in word and word != "RT"])
              for tweet in self.tweets]

    tweets = " ".join(tweets)
    if len(tweets) != 0:
        # Create the canvas to put the wordcloud
        self.canvas = FigureCanvas(Figure(figsize=(5, 3), frameon=False))
        self.axes = self.canvas.figure.add_subplot(frameon=False, xticks=[], yticks=[])

        # Create the wordcloud and add it to the canvas
        wordcloud = WordCloud(random_state=21, max_font_size=50, max_words=40, collocations=False,
                              prefer_horizontal=1, colormap="Blues", background_color="white").generate(tweets)
        self.axes.imshow(wordcloud)

        # Add the wordcloud to the tab
        self.currentDataLayout.addWidget(self.canvas, 1, 1)
```

When this is added to the tab:

This means the basic design for the tabs is complete.

Looking at the initial window design below, creation of the window has been successful.

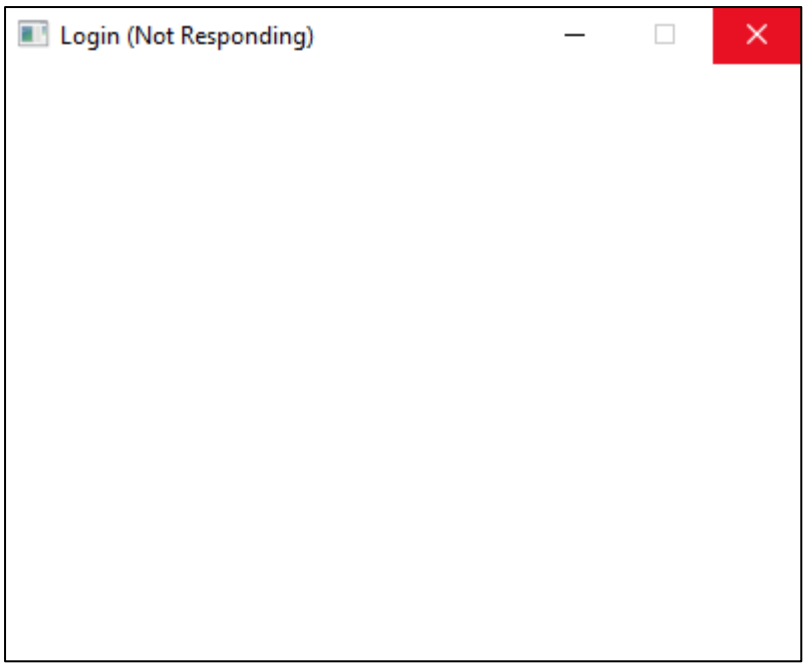


Also, when adding a new tab as demonstrated before, the new tab also follows the same basic design structure for the tab but is showing its own unique data shown by the differences in the graphs/charts.

When I add a new tab as demonstrated previously, the new tab has the same kind of structure as the dashboard but is showing different data. (The red box shows that a different tab is being viewed)

Implementing threading

When beginning to integrate the model into the software, I ran into an issue when loading the model. It would take about 15 seconds to load the model into RAM to be used by the system which should not be overly problematic and is more of just an inconvenience but this 15 seconds of loading the model would cause the system to hang, waiting for the model to be finished loading. This resulted in the screen below being shown when running the software. The main issue is that Windows interprets this 15 second of hang time as the system crashing and so forces the software to close which is problematic for obvious reasons.



To explain why this is happening, below I have shown the top of the “Model” class which was created earlier in development (there are other methods which I have not shown in the screenshot to save space). This code written within the class is outside of the constructor (“__init__”) method. This means that when the file and thus the class are imported, tools shown such as the lemmatizer and model are automatically loaded and ready to use. These tools being outside the constructor method means they are not instance specific and so only need to be loaded once which is significant as loading of the model can take over 10 seconds, even on a machine with a comparably fast SSD. The combination of the above factors means that every time the program is run the program is forced to wait over 10 seconds and therefore hang like shown above.

```
class Model:
    # Create the tools used for cleaning
    stop_words = set(stopwords.words('english'))

    slang = {
        'u': 'you',
        'r': 'are',
        'some1': 'someone',
        'yrs': 'years',
        'hrs': 'hours',
        'mins': 'minutes',
        'secs': 'seconds',
        'pls': 'please',
        'plz': 'please',
        '2morow': 'tomorrow',
        '2day': 'today',
        '4got': 'forget',
        '4gotten': 'forget',
    }

    # Create the tools used for pre-processing
    lemmatizer = WordNetLemmatizer()

    tokenizer = TweetTokenizer(reduce_len=True)

    # Load the model to be used
    model_file = open("sentiment_model.pickle", 'rb')
    model = pickle.load(model_file)
    model_file.close()
```

To fix this I will need to implement threading. This is a form of concurrency where the program will seemingly simultaneously load the model into RAM whilst also providing the user interface and other system functionality. This should mean there is no longer any freezing of the UI.

Below I have adjusted the class to have a new class method which loads the model. This means the model variable still does not belong to the instance so no instantiation of the class will be necessary.

```
class Model:
    # Create the tools used for cleaning
    stop_words = set(stopwords.words('english'))

    slang = {
        'u': 'you',
        'r': 'are',
        'some1': 'someone',
        'yrs': 'years',
        'hrs': 'hours',
        'mins': 'minutes',
        'secs': 'seconds',
        'pls': 'please',
        'plz': 'please',
        '2morow': 'tomorrow',
        '2day': 'today',
        '4got': 'forget',
        '4gotten': 'forget',
    }

    # Create the tools used for pre-processing
    lemmatizer = WordNetLemmatizer()

    tokenizer = TweetTokenizer(reduce_len=True)

    model = None
    model_loaded = False

    @classmethod
    def load_model(cls):
        # Load the model to be used
        model_file = open("sentiment_model.pickle", 'rb')
        cls.model = pickle.load(model_file)
        model_file.close()

        model_loaded = True
```

I then made changes to the main file to accommodate for these changes and implement threading.

Main function before:

```
def main():
    app = QApplication(sys.argv)  # Starts the application
    app.setStyle('Fusion') # Set the colour theme of the windows
    userInterface = Controller()  # Create an instance of the controller
    app.exec_()  # Creates a loop, displaying the application gui until it is exited

main()
```

Main function after:

```
def main():
    app = QApplication(sys.argv)  # Starts the application
    app.setStyle('Fusion') # Set the colour theme of the windows
    userInterface = Controller()  # Create an instance of the controller
    app.exec_()  # Creates a loop, displaying the application gui until it is exited

if __name__ == "__main__":
    with ThreadPoolExecutor(max_workers=2) as executor: # Create thread pool
        executor.submit(Model.load_model) # Creates a thread to load the model
        executor.submit(main) # Create another thread that runs the application
```

Now, when I run the program, the user interface is shown as needed, and the model is loaded in background so the user can sign in with minimal interruption.

This concludes development of the system. Now I can begin testing.

Stage 5 – Reflection

What has been completed?

During this stage, I fully integrated my previously created functionality into the user interface created in the previous stage. One example of this integration is the completion of the logging in and signing up feature of the application. Now, users can navigate to the sign up screen if it is their first time using the software or use the login screen if they have an existing account.

Moreover, the main window is also shown upon logging in and is now able to display the necessary sentiment analysis data relevant to the user.

Overview of the prototype as a whole

The current solution is nearly complete and almost a good representation of the finished application. The system currently is comprised of a user login system as a well as a main application window giving the user access to sentiment analysis data on their company as well as sentiment analysis data on their chosen topics. Users have the option to add more topics to be analysed as well as update their own preferences for the software such as the number of tweets to analyse per topic. The system is now also able to run in background, performing sentiment analysis for the user while they are away from the application itself.

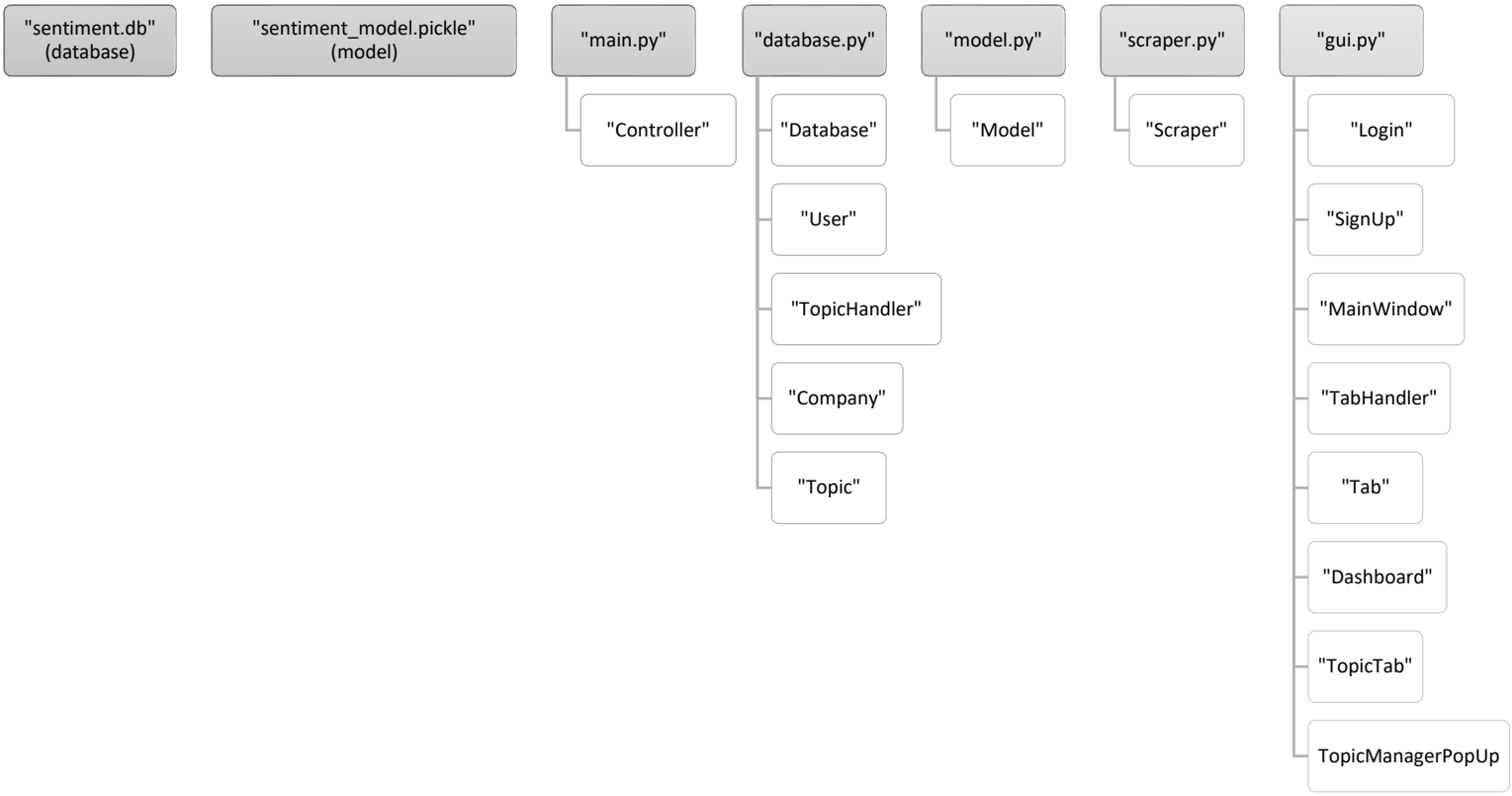
What has been tested and how?

Testing needed to be extensive during this stage as this is where most of the created functionality was finally added to the system.

In the first stage where I created the database, I tested the functions that allowed for users to register and login but I repeated these same tests again in this stage where these features were incorporated to ensure robustness of the login system. I used similar test tables as before to look at

the input validation currently in place in both the login and sign up screens, providing both valid and invalid inputs to see how the system responded. I also tested whether user details were added to the database once user had used the register screen to make an account. This was done simply by registering a test account and the running a query to see if that user record had been stored correctly in the database.

Current system structure (new items in green)



Are there any changes that may need to be made to this component further into development?

As of completion of this stage, the system is very close to completion. Any further changes to the system at this point will either be in response to the results of testing or to make refinements to the existing components of this system.

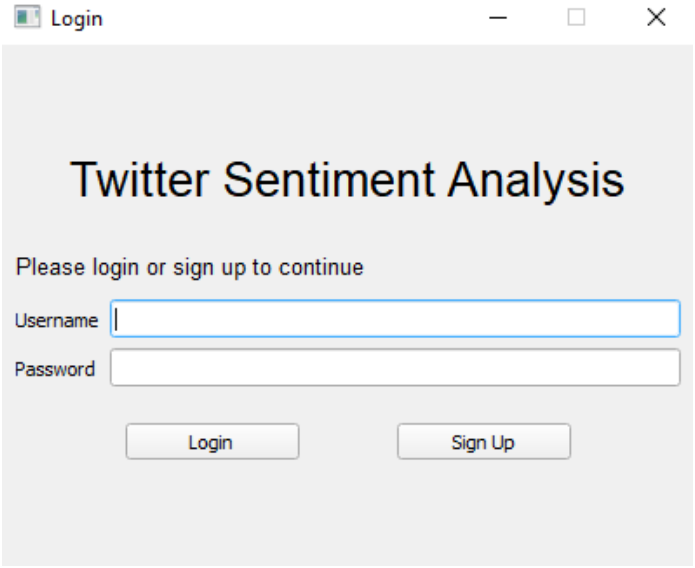
Stage 6 – Summative Testing

Test number	What is being tested?	How is it being tested?	Expected outcome
1.	UI formatting	Look for spelling errors across the program. Check UI displayed matches UI designs.	UI matches designs shown earlier in report.
2.	Login screen initialisation	Start program.	Login window is displayed correctly upon opening the software.
3.	Sign up screen initialisation	Click on “Sign up” button on login window.	Login window is closed, signing up window is displayed.
4.	Navigating back to login screen from sign up screen	Click on “Back to login” button on sign up window	Sign up window is closed, login window is displayed.
5.	Registering a user with invalid details	Entering details into sign up window input fields	Error message describing
6.	Registering a user with valid details	Entering new details on sign up window then using same details on login window to sign in	Same details entered when signing up can be used to login.
7.	Login validation	Entering invalid details into login window input fields.	Error message on window stating entered details are invalid.
8.	Main window initialisation via login screen	Click on “Login” button on login window.	Login window is closed and main window is displayed
9.	Main window initialisation via signing up window	Click on “Sign up” button on sign up window.	Sign up window is closed and main window is displayed
10.	Data is being loaded correctly	Check all widgets on the main window are displaying the data they should be.	Dashboard widgets are showing all necessary data described in design section. Tweet list is showing tweets. Graphs are showing data correctly.
11.	User topic input and UI updating	Enter new topic name into input box	New tab is added to the main window.
12.	Removing user topic	Enter topic name to be removed	
13.	Tabs showing user topics are displayed as well as dashboard	Check that every topic the user has requested to analyse is present in the tabs shown near the top of the screen	Tabs for each of the user’s topics are present
14.	Switching between tabs	Check that you can click on different tabs with each tab showing all the data relevant to that topic without errors when switching back to previous tabs.	All tabs can be clicked on to be accessed with each tab showing it relevant data with no errors.
15.	File menu.	Check each element of the drop-down menu at the top of the screen to ensure every option described in the design section is present. Check each available option in the drop-down	Each option on the file menu performs its necessary function.
16.			

Test 1 (UI formatting)

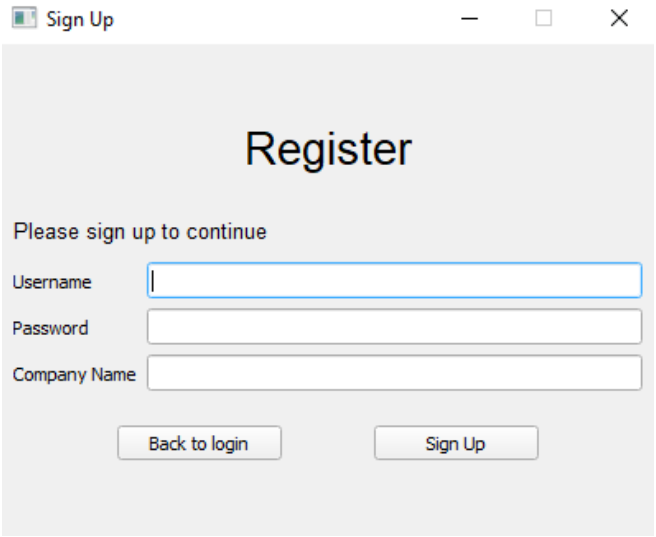
For this test I will show a screenshot of every single window that can be opened during running of the window and check for correct spelling and other grammar.

Login Screen:



Formatting is fine.

Sign up screen:



Formatting is fine.

Dashboard screen:

Formatting is fine.

Change topics screen:

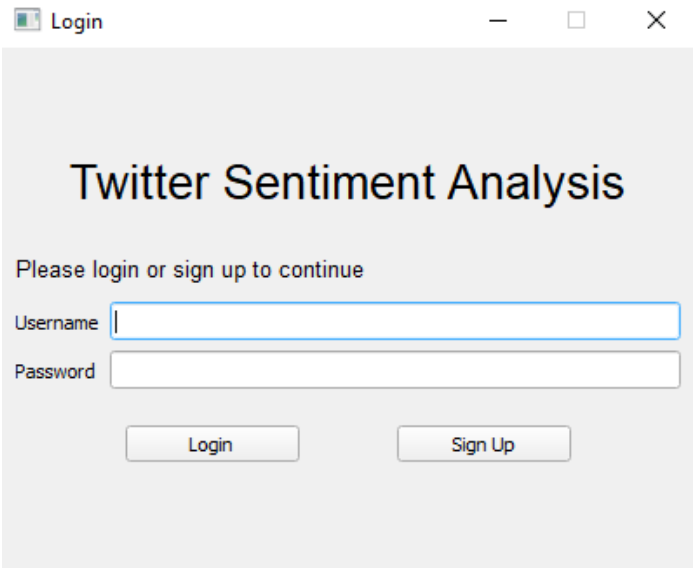
Formatting is fine.

Settings screen:

Formatting is fine.

Test 2 (Login Screen Initialisation)

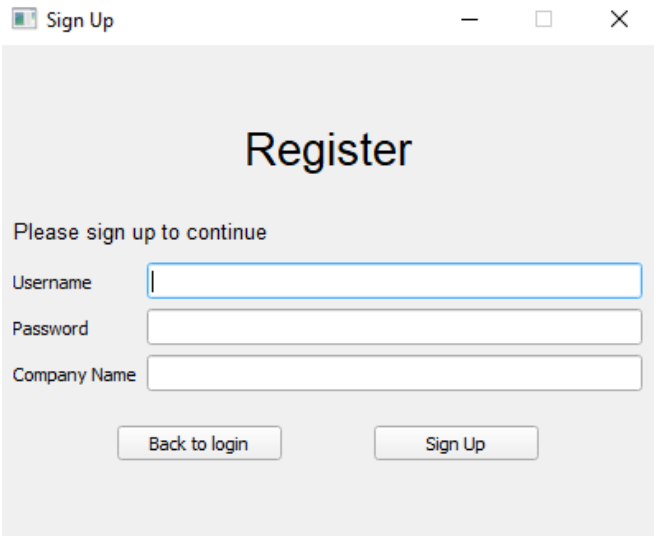
What is immediately shown when the program is start:



This shows login screen initialisation to be successful.

Test 3 (Sign up Screen Initialisation)

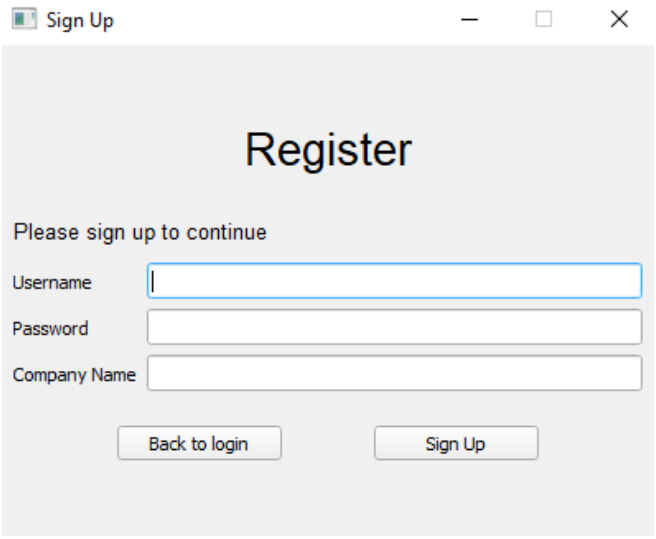
After pressing the “Sign up”:



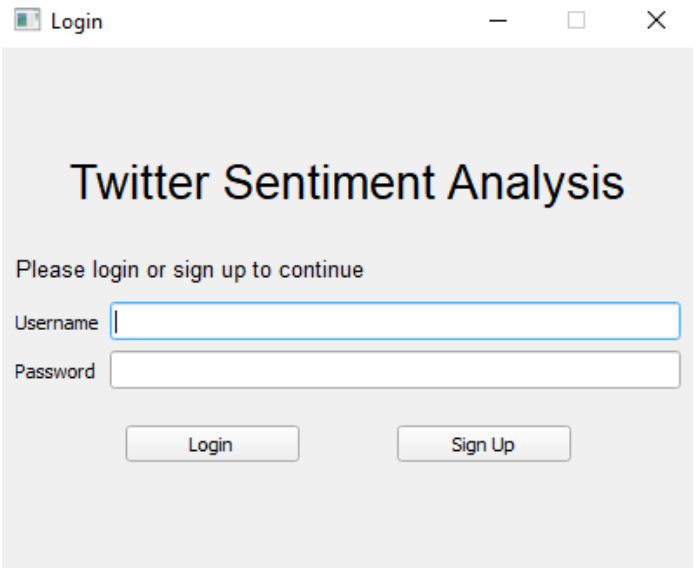
This shows sign up screen initialisation to be successful.

Test 4 (Go back to login screen)

Before clicking “Back to login”



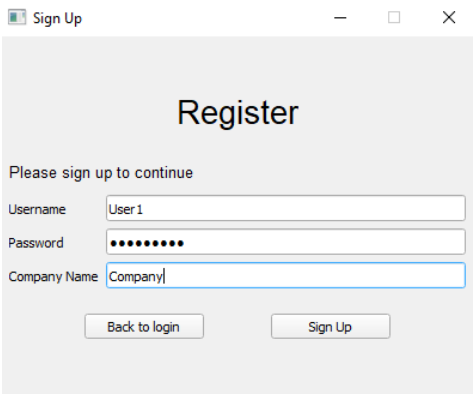
After clicking “Back to login”:



This shows the user can navigate back and forth between the login and sign-up screens

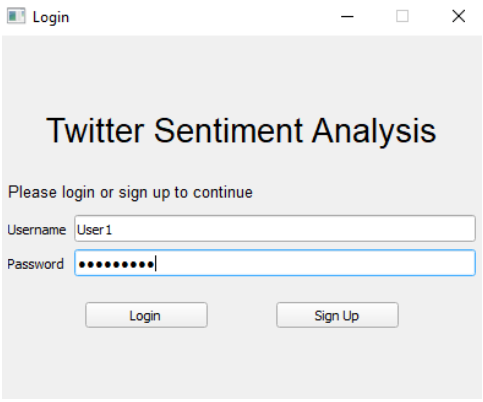
Test 4 Registering a user with valid details.

I will be creating a user with the username: “User 1”, password: “Password!” and company: “Company 1”.



After pressing “Sign Up”, this is the record added to the users table of the database:

To test the new login actually works I will now enter the same login details on the sign up screen.

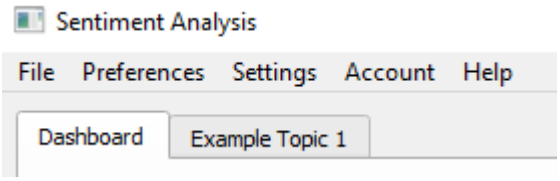


Upon pressing the “Login” button:

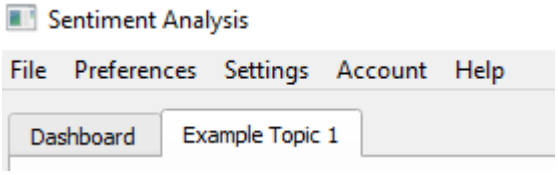
The main window is shown as required. This shows user registration and the following attempts at logging in to be possible.

Switching between tabs

Before switching tabs, the user is initially on the dashboard tab, as shown below.



After clicking on “Example Topic 1”:



This shows the user can navigate between their different topic tabs

Switching between time scales

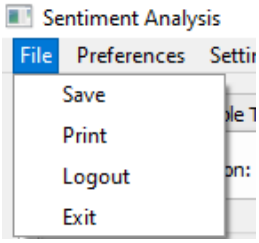
Initially, the dashboard and other tabs are showing graphs showing fluctuations in data over the last 7 days. The user has the option to switch between this 7 day time scale to show the last 8 months. To switch between time scales on the graphs, users need to press this button:

Once this button is pressed:

The graphs are now showing the data from the last 8 months instead of the last 7 days. As you can see in the screenshot below, the button has also updated to allow for the user to switch back to the 7-day time scale.

Testing the File menu

This is what the file menu looks like when opened:

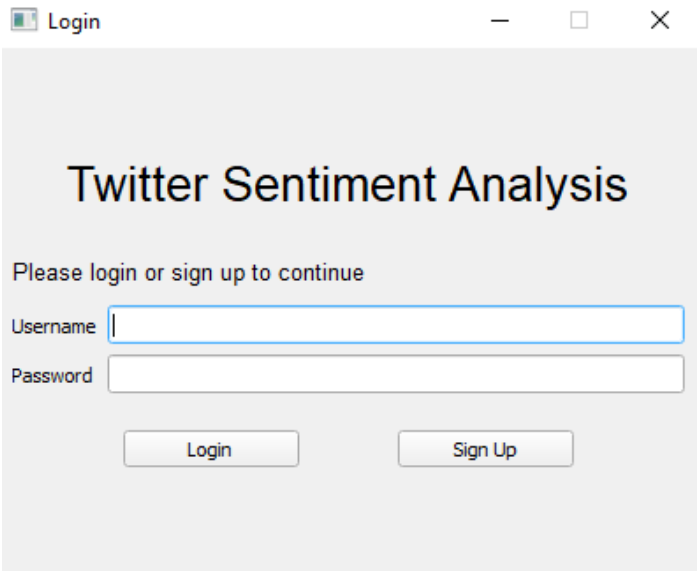


When “Save” is clicked:

When “Print” is clicked:

When “Logout” is clicked:

The main window is closed and the login window is displayed.



When “Exit” is clicked:

The main window is closed and the running of the software is stopped.

Stage 7 – Client feedback

This stage will involve my client Mr Johnson testing my system to see if it meets the required standard. I sent a client to Mr Johnson asking the following:

Questions:

1. *Were you successfully able to run the software?*
   This question is needed to establish whether basic use of the software is possible.

2. *Were you successfully able to register an account within the software?*
   This question is needed to establish whether the user could use the running software.
3. *After logging in, approximately how long did the software take to load?*
   I am mainly asking this because the model is quite large in terms of memory usage. Ideally it won't take long especially since implementing threading. The time taken to load the system is an important factor when considering the user experience as long waiting times can be frustrating for any user.
4. *Was the dashboard showing the data correctly after logging out and then logging in graphs showing correctly?*
   This is so I can see if the database access works as needed on other machines as the system was only tested on a single computer.

5. *Did you ever need to use the user guide when using the software?*
   This is so I can see if the UI is intuitive enough for the user to make sense of it without any help.

Client responses:

1. *Were you successfully able to run the software?*
   "Yes, it was easy to run the software on my own machine and there were no issues with the installation of the software. I did notice that the system was quite demanding for memory, but this wasn't an issue for my machine"

2. *Were you successfully able to register an account within the software?*

   "Yes. The login system worked as I had expected"

3. *After logging in, approximately how long did the software take to load?*
   "From what I remember, it took around 15 seconds to load the main window once I had logged in using the login window. I imagine this is for loading in all of the data from the database and updating the current data"

4. *Was the dashboard showing the data correctly, for example, were the graphs showing correctly?*
   "For both my main dashboard and my topic screens, the data was always shown as I needed it to be."
6. *Did you ever need to use the user guide when using the software?*
   "Yes, just to make sure I was using it correctly, but the software was intuitive, and the GUI was simple to use so I probably didn't even need to use the guide"

7. *Were you able to add new topic analyses tabs and have them load again when reusing the software?*
   "Yes, and it was good to see the changes in what people were saying over the last 7 days but I think that If I were to use it regularly, I would most likely only use the dashboard to see what people are saying about the company specifically."
8. *How often did you use the use the software?*

   I only used the software a few times each week just to see the changes in what people have been saying. It seems as though the software has been doing the analysis only when I

opened and logged in to the system. I think it would be helpful if there was a way the system could do these analyses automatically without me needing to login.

Response to client feedback

Looking at how the client answered my questions, he liked the software.

One issue described by the client was the software's ability to independently perform analyses in background. This is certainly something I had considered when developing the software and was an optional aspect of the success criteria. The main reason I didn't implement this is because it could have adverse effects on the running on the system due to the size of the model that would be loaded in every time the user logged in. As well as this, it was also important to consider the limitations of the access to Tweets provided by the Twitter API. Using logs in this way would mean that the system would have to make logs for every company and every topic of every user stored on the system as often as every day. This would possibly mean the retrieval of 10s of thousands of tweets each day which would likely exceed the amount permitted by the API and would therefore lead to inconsistencies in the data being provided for the user.

Evaluation

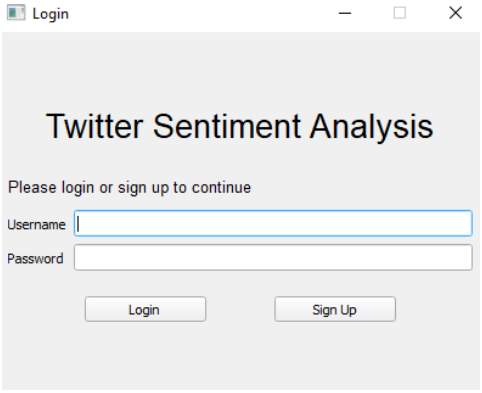
Success Criteria

Aesthetics

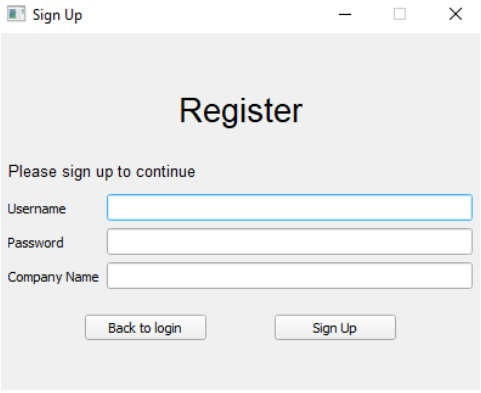
Requirement	How is it measurable?	Is it essential?	Completed?
The software will have a professional look with consistencies in font style and font size large enough to be easily readable	A screenshot of the window while the program is running will be provided.	✓	✓

This is a subjective requirement, but I still think that the software looks professional and is consistent in design, evident when comparing the login window and signing up window:

Login window:



Sign Up:



Requirement	How is it measurable?	Is it essential?	Completed?
The windows for the user interface will be large enough to contain any visual elements that the user would need to see.	A screenshot of the dashboard with all relevant data about the company is to be provided as well as a screenshot of a window showing analyses of a user-requested topic.	✓	✓

The intention with this criterion is so that data is well presented and so easy for the user to understand and use. Below is a screenshot of the graph showing sentiment fluctuations for a topic over the last 7 days compared to the logs the system made containing this data. This shows that the data is being presented in a way the prioritises readability and usefulness.

:

I have made all of the graphs very large so that the user can easily understand the data they are showing. Moreover, the graphs all have high contrasting colours so they are easy to read. This is clear in the screenshot below which shows what

Requirement	How is it measurable?	Is it essential?	Completed?
The user will be able to change the colour scheme of the user interface to match their preferences.	Two screenshots of the window will be provided, each with different colour schemes.	x	x

I decided not to include this option for the software due to the limitations of the library used for the user interface which did not allow for the consistent change of colour themes within the UI.

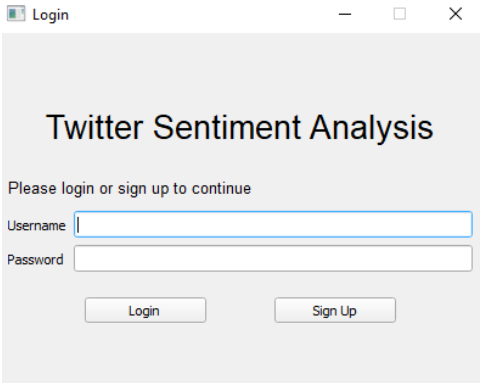
Requirement	How is it measurable?	Is it essential?	Completed?
The software needs to have a simplistic design so that it is not difficult to use.	A screenshot of the main window with large readable elements is to be provided.	✓	✓

The software is very easy to use and requires almost no user input as it is mostly automated. The only potentially difficult thing to do would be for the user to add more topics to be analysed but this is fully explained in the user guide which has been made to be easy to find.

Outputs

Requirement	How is it measurable?	Is it essential?	Completed?
A login/sign up window is provided prompting the user to enter their details	A screenshot of the login/sign up window will be provided.	✓	✓

Below is the login window provided when the user initially runs the software:



Requirement	How is it measurable?	Is it essential?	Completed?
The main window has a dashboard containing analysis on the user's company	A screenshot of the dashboard will be provided.	✓	✓

Below is the dashboard provided for a company with the name "Intel".

Requirement	How is it measurable?	Is it essential?	Completed?
Sentiment analysis data will be called from a database to show sentiment change over time.	A screenshot of a graph showing changes in trends over time will be provided.	✓	✓

Below are the time series graphs which show the changes in data over time. The graphs shown are about a company called “Intel” and the data being shown was collected by the system in the months and days leading up to the date when the screenshot was taken.

Requirement	How is it measurable?	Is it essential?	Completed?
There will be a help screen where users can find out how to use the software and find out more about the software	A screenshot of the help screen will be shown	✓	✓

I was able to add a help section on the menu bar which contained an about window for the user to find out about the software and a user guide to ensure the user knows how to navigate and make best use of the software.

Below is the “About” window:

Inputs

Requirement	How is it measurable?	Is it essential?	Completed?
The user will be able to change the type of graph representing some of the on-screen analyses. E.g change a bar chart to a pie chart or vice versa.	Two screenshots will be provided, each with a different graph type representing the same data.	✓	✓

Requirement	How is it measurable?	Is it essential?	Completed?
There will be a settings menu where the user can adjust the software settings	A screenshot of the settings menu will be provided	✓	✓

Below is the settings menu presented for the user when selecting the settings option on the menu bar.

Requirement	How is it measurable?	Is it essential?	Completed?
Where appropriate, data will be shown graphically.	A screenshot of the embedded graphs on the main window will be provided.	✓	✓

This criterion was very successful in its implementation. I was able to add multiple graphs/ charts with several being able to dynamically switch between two different time scales that the user might want to use to look at data changes over time. Below I have provided a screenshot of each of the graphs/charts shown on the main window.

Line graph showing change in average sentiment over time:

Line graph showing change in number of tweets abouts the company/topic over time:

Word cloud showing common key words from the tweets most recently scraped:

Usability features

One example of a usability feature I have added to the final software is threading. This allows the model needed to make tweet predictions in the background while the user is navigating the login/ signup screen. Threading here means that the UI and model are loaded concurrently, which means the UI doesn't freeze when the model is being loaded, improving the usability of the solution.

Another usability feature I have added is creating an easily readable user interface. The user interface makes use of high contrasting colours as well as large font sizes where possible to ensure that the user is able to understand what is happening on screen. This is most obvious with the graphs shown on screen as these are the most important things for the user to be able to view. All of the graphs have bright colours which stand out for the user and will help those who may otherwise find them hard to read.

Another simple usability feature I have made sure to make use of is clear and simple navigation of the user interface. Easy navigation often isn't overly noticeable when present but adds a lot to the user experience, generally making the software significantly easy to use. One example where I have tried to make navigation very simple is the main user dashboard as this is where the user will spend most of their time using the software. A specific example on the dashboard is the user's option to switch between time scales on the graphs showing the sentiment data. This switching between time scales could have been handled easily by simply opening a new window, showing the necessary graphs the user wishes to see. Instead of this, I added a button in the corner of the dashboard which when clicked dynamically switches the graphs on the window to the different time scale i.e last 7 days to last 8 months.

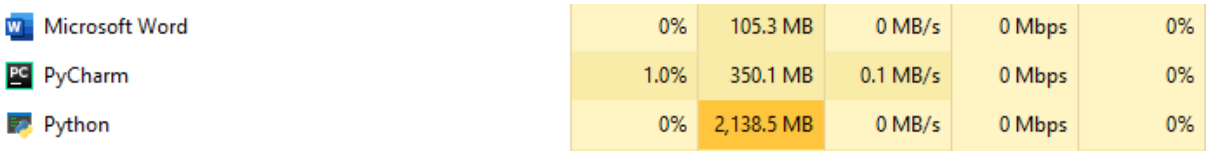
Limitations

Limitations of the current classifying model

As expected, one of the biggest limitations of the software is the model used to perform sentiment analysis. There are a few ways the problem of an inaccurate model can be amended. The most obvious approach to this issue is just creating a new model and training until sufficient accuracy and simply replacing the existing model with this new version. The type of model used was a simple Bayesian binary classifier meaning it could only produce two possible outputs however there is also the options for more advanced NLP deep learning models which are able to recognise the

relationships between words and phrases in text as well as just identifying key words which impact sentiment.

Looking at the screenshot below, taken from windows task manager, the less complex Bayesian model used in the system still consumes a significant amount of system memory. A more advanced model, would undoubtedly require more system memory due to the size of such models and the pre-processing tools required to make use of them. This means, that for machines with lower memory capacity, the current system with the Bayesian model appears to be a more appropriate option. Perhaps to address this issue, in future different versions of he software could be made where each is uses a different classifying model so the user can evaluate the trade-off between accuracy of the data and the performance of the running system.



Limitations of tweet selection and improving data retrieval approach

In general, the model is successful at handling actual opinion tweets from general twitter users, however, when retrieving tweets, the scraper can struggle differentiating the more relevant and useful tweets from those which don't offer any opinion. This becomes problematic when the model makes predictions on these irrelevant tweets and falsely lists them as being negative tweets aimed at the company which harms the reliability of the system's results.

To mitigate this issue, a different API could be used that offers greater refinement when retrieval of tweets as well as implementing some filtering that ensures that only relevant tweets are retrieved. An easier fix which coincides with the previous issue is making use of a different model which can recognise tweets that aren't offering an opinion and are a simple statement with no intended sentiment.

Limitations of the Twitter API and mitigation

Another limitation I encountered with the system during development is the limited access to tweets using the Twitter API. As discussed before, the API only allows for a limited number of tweets to be scraped in a given time period. Before development, I stated this would not be an issue for the system due to the infrequency of tweet requests, but this ended up becoming a problem when testing a user account with a large number of topics to be analysed which required thousands of tweets to be requested each time a log was to be made. After the limit to the number of tweets to be requested is exceeded, the API will respond with an error and there will be a cooldown, so the system has no other option than to tell the user that they need to wait for some time for the cooldown.

Maintenance of the system

Maintainability of the system is how easy it is to upkeep which includes amending errors as they appear. Good maintainability means that in future, other developers can easily make the necessary changes to update and correct the design of the system.

One way I have made my system maintainable is the use of modularity throughout. This is beneficial because the use of separate files and dedicated classes will make it easier to navigate to the

necessary part of the code that requires updating. It also means that changes to one part of the program should not have adverse effects on other areas of the program. This use of modularity will also prove beneficial if any further development is to take place. For example, the existing class structure within the class would make it an easy process to add in new features such as new windows or implement a different classifying model.

Further modularity within the system is the choice of libraries where I have opted to use popular and well-documented libraries like PyQt5 so that developers that may want to add to the code in future or fix any bugs will easily be able to access the tools that would allow them to do so.

I have also, where appropriate, included comments in the written code. This will aid developers that are unfamiliar with the code to understand what is happening and therefore be capable of maintaining and adding new features to the code.

Further development of the solution

A more straightforward addition to the software would be support for more platforms. Currently, the system only looks at twitter data but it could be valuable for the users if the system also considered other social media sites as well as potentially looking at review sites. Given the modularity of the system as described in the previous section, the addition of more sites would be straightforward. Currently, there is a class in the scraper file for accessing Twitter data, addition of other sites could simply mean adding a new class for each new site to be added and then making some minor tweaks to other parts of the software to make best of these other sites.

Although the model is providing adequate results, further development could involve the creation of a better sentiment analysis model for use in the system. Further work on the algorithm and changes in approach in its creation could result in greater accuracy in predictions which would be of great benefit for informing the users. One way of doing this could be using a deep learning model instead of a more conventional machine learning classifier used in the current version. Deep learning models are a more complex but are an often more accurate approach to sentiment analysis. This is because they are often able to look at all the words used in text as a whole and consider the relationships between these words and therefore form a more complete understanding of the text it is analysing. This addition will become easier to implement with time as access to deep learning becomes easier.

Bibliography

Source title	Author/ Publisher	Source URL	Date accessed	Use of source
Sentiment Analysis – MonkeyLearn	MonkeyLearn	https://monkeylearn.com/sentiment-analysis/	27/06/20	Sentiment analysis research, understanding basic concepts
Opinion mining and sentiment analysis	Bo Pang and Lillian Lee	https://www.cs.cornell.edu/home/llee/omsa/omsa.pdf	27/06/20	Sentiment analysis research, understanding more complex concepts
Deep learning vs machine learning: a simple way to understand the difference	Zendesk	https://www.zendesk.co.uk/blog/machine-learning-and-deep-learning/#:~:text=To%20recap%20the%20differences%20between,intelligent%20decisions%20on%20its%20own	27/06/20	Understanding the difference between machine learning and deep learning
Sentiment Analysis with Machine Learning: Process & Tutorial	MonkeyLearn	https://monkeylearn.com/blog/sentiment-analysis-machine-learning/	27/06/20	Deep learning and machine learning research
All-in-One Text Analysis & Data Visualization Studio	MonkeyLearn	https://monkeylearn.com/monkeylearn-studio/	28/06/20	Product research for Monkey Learn
Text Analytics	Microsoft	https://azure.microsoft.com/en-gb/services/cognitive-services/text-analytics/	28/06/20	Product research for Microsoft Azure
Brandwatch By Use Case	Brandwatch	https://www.brandwatch.com/use-cases/	28/06/20	Product research for Brandwatch
How to Scrape Tweets From Twitter	Martin Beck / towards data science	https://towardsdatascience.com/how-to-scrape-tweets-from-twitter-59287e20f0f1	11/07/20	Use of limitations of the standard Twitter API
SENTIMENT ANALYSIS – THE LEXICON BASED APPROACH	Alphabold	https://alphabold.com/sentiment-analysis-the-lexicon-based-approach/	20/07/20	Understanding the lexicon approach to sentiment analysis
FLASK SQlite Python - Checking if username is already in database	StackOverflow	https://stackoverflow.com/questions/40205518/flask-sqlite-python-checking-if-username-is-already-in-database/40205914		Database access
How autoincrement works in sqlite	DatabaseGuide	https://database.guide/how-autoincrement-works-in-sqlite/		Researching database autoincrement

python + sqlite, insert data from variables into table	StackOverflow	https://stackoverflow.com/questions/4360593/python-sqlite-insert-data-from-variables-into-table		Inserting data from variables
Encrypting passwords for use with Python and SQL Server, *ImportError: DLL load failed: The operating system cannot run %1*	Burt King, Github	https://www.mssqltips.com/sqlservertip/5173/encrypting-passwords-for-use-with-python-and-sql-server/ https://github.com/pyca/cryptography/issues/4011		Password encryption
Getting today's date in YYYY-MM-DD in Python?	StackOverflow	https://stackoverflow.com/questions/32490629/getting-todays-date-in-yyyy-mm-dd-in-python		Getting today's date
SQLite Update, python sqlite3 UPDATE set from variables	Sqlitetutorial, stackoverflow	https://www.sqlitetutorial.net/sqlite-update/ https://stackoverflow.com/questions/39372932/python-sqlite3-update-set-from-variables		Updating database tables
Sentiment Analysis with TensorFlow 2 and Keras using Python	GeeksforGeeks	https://www.geeksforgeeks.org/python-stemming-words-with-nltk/		Cleaning data
Basic Tweet Preprocessing in Python	Parthvi Shah/ towardsdatascience	https://towardsdatascience.com/basic-tweet-preprocessing-in-python-efd8360d529e		Cleaning data
How to rearrange Pandas column sequence?	freddygv/ StackOverflow	https://stackoverflow.com/questions/12329853/how-to-rearrange-pandas-column-sequence/23741704		Cleaning data
Twitter Sentiment Analysis – Classical Approach VS Deep Learning	Kaggle	https://www.kaggle.com/josephassaker/intro-to-deep-learning-sentiment-classification		Model creation
Creating additional windows	Martin Fitzpatrick / LearnPyQt	https://www.learnpyqt.com/tutorials/creating-multiple-windows/		Window creation
PyQt5 – Set fix window size for height or width	GeeksforGeeks	https://www.geeksforgeeks.org/pyqt5-set-fix-window-size-for-height-or-width/		
Menus and toolbars in PyQt5	ZetCode	http://zetcode.com/gui/pyqt5/menustoolbars/		Window menu
Creating Menus, Toolbars, and Status Bars	Leodanis Pozo Ramos / Real Python	https://realpython.com/python-menus-toolbars/		Window Menu

PyQt5 Menubar	TechWithTim	https://www.techwithtim.net/tutorials/pyqt5-tutorial/menubar/		Window Menu
PyQt5 QMessagebox	TechWithTim	https://www.techwithtim.net/tutorials/pyqt5-tutorial/messageboxes/		Creating the tab manager popup
PyQt5 input dialog	Python Basics	https://pythonbasics.org/pyqt-input-dialog/		Creating