Technical Guide

Sentiscape

CA400

Jason Boylan: 18342986
Kelan Smyth: 18342973

Dr. Hyowon Lee

Date Completed: 24/04/2022

Table of Contents

Abstract

Sentiscape is a mobile messaging application designed for Android smartphones. It is aimed at users looking for an application that can convey the affective side of messages as well as the informative side. It will also make it easy for users to categorise and find past conversations, using natural language processing and sentiment analysis to create emotive summaries of conversations. It is entitled Sentiscape, thanks to its association with sentimentalism.

Overview

The purpose of the system is to create a mobile messaging application for users looking for an application that can convey the affective side of messages as well as the informative side. It will also make it easy for users to categorise and find past conversations, using natural language processing and sentiment analysis to create emotive summaries of conversations.
It is entitled Sentiscape, thanks to its association with sentimentalism.

Sentiscape will allow users to perform the basic functions of any messaging app, such as sending messages, attaching multimedia and creating group chats.

Also with Sentiscape, the app will re-think how text-messaging interaction can be done and subsequently, design novel user interfaces by bringing in up-to-date computational technologies at the back-end of the app.

This will be displayed with primarily two major changes. Firstly, the application will allow users to categorise their conversations with friends which will make it easier to look for past conversations. Conversations will be broken down by day and a summary of the conversation will be created to allow users to easily remember what conversation took place on a particular date.

Secondly, it will track the mood of the conversation which will allow for different emoticons and colours depending on the mood. This will be done using sentiment analysis to analyse the mood of the conversation. If the most prominent emotion being displayed is sad, there will be more muted colours and suggested emoticons will relate to the conversation. Chat bubbles will be different colours depending on the mood.

Glossary

IDE: An Integrated Development Environment is a type of software application that allows users to create applications using comprehensive developer tools. It consolidates different tools such as a text editor and a debugger.

JSON: JavaScript Object Notation is an open standard file format and data interchange format that uses human-readable text to store and transmit data objects consisting of attribute-value pairs and arrays. It is a common data format with diverse uses in electronic data interchange, including that of web applications with servers.

API: An Application Programming Interface is defined as a linking connector between different computer systems or programs.

Android Studio: Android Studio is the official integrated development environment for Google's Android operating system, built on JetBrains' IntelliJ IDEA software and designed specifically for Android development. It supports the programming languages Java, Kotlin and C++.

SQLite: SQLite is a database management system that uses SQL (Structured Query Language).

NoSQL: NoSQL databases are databases that are non-relational and can store large amounts of unstructured data.

Firebase: Firebase is a platform designed by Google that allows its users to develop web and mobile applications. It allows for account management systems, such as account creation and deletion.

Cloud Firestore: Cloud Firestore is a scalable NoSQL database for mobile, web, and server development from Firebase.

Jython: Jython is a Java implementation of the Python programming language. It allows Python to run on Java.

Chaquopy: Chaquopy is a software development kit that allows users to integrate Python scripts with Android Studio.

UID: Unique Identification Number that is generated for each user.

Natural Language Processing: Natural Language Processing (NLP) refers to the field of computer science and artificial intelligence that studies the interaction between computers and the human language. It focuses on applying linguistic and statistical algorithms to text in order to improve communication between computers and humans.

Text Summarisation: Text summarisation is an NLP technique that is used to summarise information in large texts for faster reading.

Sentiment Analysis: Sentiment analysis is an NLP technique that is used to detect whether the meaning behind a message is positive, negative or neutral.

Recursive Neural Tensor Network: Recursive neural network with a tree structure containing a neural net at each node.

Conversation Summary Feature: The Conversation Summary feature is the feature that summarises users' conversations using natural language processing.

Affection Feature: The Affection feature is the feature that provides different colours and emoticons based on the emotion felt in the user's messages. This is accomplished through sentiment analysis.

SMS: Short Service Message is a standardised text messaging service in most mobile telecommunications systems.

HTTP: Hypertext Transfer Protocol is designed to enable a request-response protocol between a client and a server.

POST Request: A POST request is an HTTP method used to send data to a server.

GET Request: A GET request is an HTTP method used to request data from a server.

XML: Extensible Markup Language is a markup language that mainly focuses on the transfer of data.

SVG: Scalable Vector Assets is an XML-based image asset used for 2D graphics that allows for scalability without a deterioration in quality.

Motivation

The motivation for this application came from a few different places. Firstly, as part of our project last year, we worked on a system that incorporated some machine learning algorithms and we wanted to work in the same field again. This was the reasoning behind our push for natural language processing and sentiment analysis to be a crucial part of the application.

Secondly, we noticed various shortcomings of existing messaging applications, such as the difficulty in accessing past conversations and the challenge that existed when trying to get across the affectionate side of a message.

Thirdly, we were inspired by Kansei engineering, a Japanese term that translates to "emotional" or "affective" engineering. This involves translating the consumer's feelings into parameters that can be used as part of a product/service. We tried to incorporate some of

these ideas into our application, particularly when it came to how users message each other. We read some different articles on this for some ideas [1]. This influenced the colour palette of the application.

Research

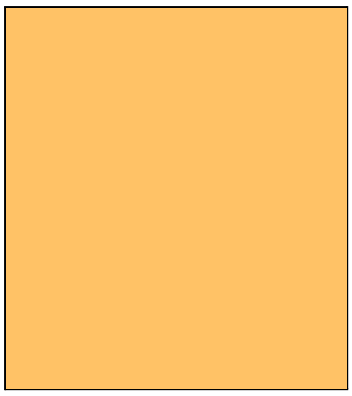
Deciding on Colours

Deciding what colours to use when creating the application was an important part of our research. We had to make sure the colours chosen for specific emotions matched the emotion itself. To make sure we chose colours that were appropriate, we conducted research to allow us to make a fully informed decision. We studied colour psychology [2] and came to the decision that a positive message should be accompanied by a muted orange tone. Yellow can be positive but can also represent warning or danger, so we leaned towards orange which more closely represents “happiness” and “enthusiasm”. Too strong a shade of orange would be too “attention-grabbing”, so we settled for a muted tone [3].

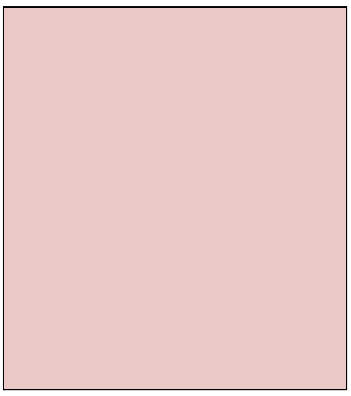
As for negative feelings, we decided on a teal shade of blue that would best represent the emotion. Although the colour blue can represent “stability” and “calmness”, it was more prominently felt to be “distant” and “icy” and represents “sadness” [4].

The colour for neutral was chosen as a light pink for two reasons: firstly, it was clearly visible in both light and dark mode. Secondly, it fit with the general colour scheme of the application.

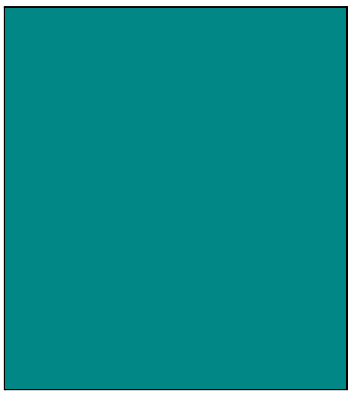
We then made use of colour pickers to assist us in choosing the correct colour for our application. We tested a variety to see what looked best in the application. These were the colours we chose:



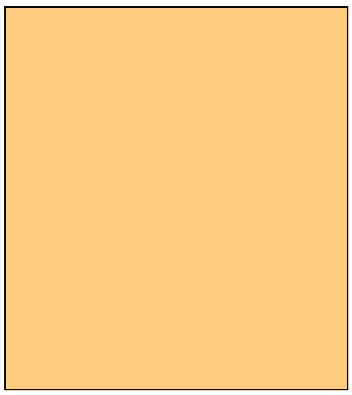
#FFC266
Positive (Receiver)



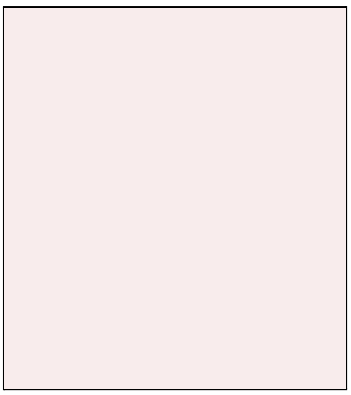
#EAC8C8
Neutral (Receiver)



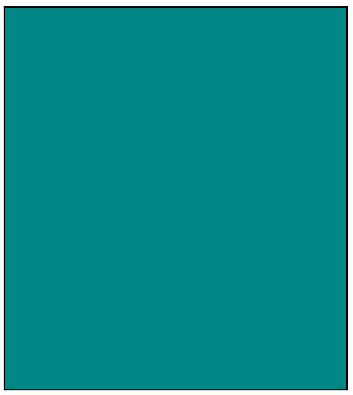
#008080
Negative (Receiver)



#FFC80
Positive (Sender)



#F8ECEC
Neutral (Sender)



#FF018786
Negative (Sender)

Slightly different shades were used for the sender and receiver message bubbles in order to make it as clear as possible who sent the message.

Deciding on software and hardware

It was decided to develop the application for Android for two reasons: firstly, we're both familiar with Java which Android Studio supports. Secondly, we both had access to Android smartphones which helped make testing the application easier and quicker than using an emulator.

We chose to use Android Studio as our IDE because it was the clear and obvious choice. It has lots of documentation, is regularly updated and is officially supported.

This let us develop the application in Java but also allowed us to include XML for the layout of the application.

Python was also used for some natural language processing as we had experienced using Python's machine learning modules in our third-year project. We felt that these libraries were well-developed, well-documented and relatively easy to work with.

Java versus Kotlin

When we decided on using Android Studio as our application, we had to make a decision on which programming language we'd use. Android supports both Java and Kotlin, which both come with advantages and disadvantages. We researched comparisons between both before coming to a decision [5].

Java was a language we were both familiar with for a couple of years already, whereas we had no experience with Kotlin. On the other hand, Kotlin is more "concise and expressive" and is more user-friendly.

In the end, we chose Java due to our experience with it and there was more documentation in that language as it has been around for a lot longer (1995 vs. 2016).

Datasets and Existing Models

We looked at various datasets to enable us to train and test our machine learning aspect. Our project focuses on the subject of natural language processing. This can be further broken down into two sections: text summarisation and sentiment analysis.

Text Summarisation

The first step in implementing a working Summarisation feature was to choose which type of text summarisation to perform: extractive summarisation or abstractive summarisation. Extractive summarisation involves retrieving the most important and meaningful text and returning it as a summary. Abstractive summarisation involves returning a different text to the input and creating a new sentence(s) as the output. We chose to prioritise abstractive summarisation.

Next up was to find a suitable dataset to train our model on. First of all, we looked at the NPS Chat Corpus [6]. We began with this one first because of the data it contained. It was made up of “10,567 posts out of approximately 500,000 posts [they] have gathered from various online chat services in accordance with their terms of service”. We thought this was relevant as the content of the dataset was made up of conversations between users in online chat forums. This bore a resemblance to our application and as a result, we began working with it.

One issue we had when working with this dataset was that the messages tended to be a lot longer than those typically found in a text messaging service and the messages weren’t typically between just two people.

Next, we came across another corpus that was more in line with our needs. This was the SAMSUM corpus dataset [7]. The SAMSUM corpus was more appropriate for our project than the NPS Chat corpus because the dataset consisted of over 16,000 conversations between two people. This was more appropriate for a messaging application, and the messages tended to be shorter. For example:

```
{
      'id: '13818513',
      'summary': 'Amanda baked cookies and will bring Jerry some
tomorrow.',
      'dialogue': "Amanda: I baked cookies. do you want some?\r\nJerry:
Sure!\r\nAmanda: I'll bring you tomorrow :-)"
}
```

These messages were similar to conversations we felt would happen in *Sentiscape*. While investigating different methods for implementing the SAMSUM dataset, we came across a

library known as Transformers by Hugging Face [8]. This library included pre-trained datasets with which we could incorporate our data and as part of our research, we tested how this pre-trained model worked using a number of conversations we had had on Discord (a messaging platform used frequently when working on the project). Here is an example of one of these full conversations:

```
Kelan:
I have Firebase working btw so shouldn't take too long to catch up to
where we were and have messaging implemented.

Jason:
Brilliant

Kelan:
Probably should've said that a couple days ago lol.
Just slipped my mind.

Jason:
did you commit it?

Kelan:
No not yet, I will this evening before I finish up.

Jason
Grand.

Kelan:
So any accounts you make on sign up are permanent now so we should stick
to only using a couple.
And to check the firebase console you'll have to log into my DCU account.
which I'll text you at some stage.

Kelan:
Pushed.

Jason:
grand thanks.
```

Here is the summary generated for this conversation:

```
Kelan: I have firebase working btw so shouldn’t take too long to catch up.
Jason: did you commit it? Kelan: No, I will this evening before I finish
up.
```

This summary was generated using a base model which didn’t use the SAMSUM corpus as its dataset. By reviewing the different models available on the HuggingFace website, we found some models that were trained on the SAMSUM corpus and tested those out instead.

In the end, the model `bart-large-samsum` by Linydub was chosen. This model was trained with the SAMSUM dataset and returned accurate results.

By reviewing a number of these summarisations, we were satisfied with the results and pivoted away from training our own dataset and towards using a pre-trained model.

Sentiment Analysis

The first step in deciding what machine learning algorithm to use for sentiment classification was to research what options were available. Originally the aim was to incorporate the python nltk library into the application via Jython to implement sentiment analysis, however this idea was later abandoned as Jython could not support the most recent versions of the library.

The Stanford CoreNLP library was later decided upon as it did not require older versions in order to run the sentiment analysis in the application. The Stanford CoreNLP sentiment analysis model was trained using a recursive neural tensor network on a movie review dataset. The model parses the data into sentiment trees where each word in the text is placed on a leaf node and represented as a vector. The model does not produce a final numerical score, instead it categorises the text as “negative”, “neutral”, “positive”, or “very positive”. For our use of the model we decided to simplify the classification into “negative”, “neutral”, and “positive” by including “very positive” results in the “positive” categorisation.

Since the model was trained using a movie review dataset we researched training the model on a custom dataset to refine the results returned in the application. However since the model used a unique classification system it was difficult to find a suitable dataset without having to modify thousands of values to account for this so the standard model was deemed satisfactory due to movie reviews being relatively well structured and very opinionated.

Deciding which libraries to use

When researching how we’d create our application, we realised we would have to incorporate a lot of external libraries into our system in order to help us perform tasks. Here, they are broken down by feature.

Summarisation

As we were using Python to implement the Summarisation feature, we had to find some libraries that would allow us to perform machine learning operations on our data. Python supports a lot of libraries to assist us in this, such as PyTorch, Transformers and Tokenizers. As part of our research, we chose to place an emphasis on using Transformers as our main library. Transformers makes use of Tokenizers and PyTorch when running some of its functions.

Sentiment Analysis

The Stanford Core NLP library [9] was implemented in the application, which contained a wide array of natural language processing tools but most importantly it supported sentiment analysis. An important factor of this choice was the model's ability to process data at an adequate speed to suit the needs of an instant messaging application. Originally the plan was to have the sentiment analysis model run locally on the android device but it was later decided that this was not the best way to incorporate the functionality into the application.

We tested the accuracy of the Stanford CoreNLP model by creating an array of standard messages and recorded what the predicted sentiment output should be and comparing it to the actual output of the model. From researching sentiment analysis models it is clear that current day models leave a lot to be desired, but our results returned adequate results showing an accuracy rating of around 70% for positive messages and 60% for negative messages.

Incorporating the model into the Android device would require it to handle the processing of the data which would put a heavy load on the CPU and cause the application to run slower on older devices and potentially be unsupported by devices with an older Android SDK. To bypass the CPU usage and hardware requirements, Stanford CoreNLP can be hosted on a dedicated server that allows devices to send POST requests containing the data they want to be analysed and returns the sentiment classification.

```
kelan@HP-Notebook:~$ bash startserver.sh
[main] INFO CoreNLP - --- StanfordCoreNLPServer#main() called ---
[main] INFO CoreNLP - Server default properties:
                        (Note: unspecified annotator properties are English defa
ults)
                        annotators = sentiment
                        inputFormat = text
                        outputFormat = json
                        prettyPrint = false
[main] INFO CoreNLP - Threads: 4
[main] INFO CoreNLP - Starting server...
[main] INFO CoreNLP - StanfordCoreNLPServer listening at /0:0:0:0:0:0:0:0:9000
```

Stanford CoreNLP server

```
ngrok by @inconshreveable                                                    (Ctrl+C to quit)

Session Status                online
Account                       kalitest2121@gmail.com (Plan: Free)
Version                       2.3.40
Region                        United States (us)
Web Interface                 http://127.0.0.1:4040
Forwarding                    http://14ca-2001-bb6-4898-7a58-44e6-3b7-e276-427c.ngrok.io -> http://localhost:9000
Forwarding                    https://14ca-2001-bb6-4898-7a58-44e6-3b7-e276-427c.ngrok.io -> http://localhost:9000

Connections                   ttl     opn     rt1     rt5     p50     p90
                              0       0       0.00    0.00    0.00    0.00
```

Ngrok Tunnel

By hosting the server on a local machine and creating a tunnel using Ngrok [10] to allow devices to contact the sentiment analysis service, we created an application that could be run on the majority of android devices and make use of a sentiment analysis model that

could process data quickly to suit the requirements of an instant messaging application. Response time for an average message (60 characters) was roughly 1.426 seconds.

Android Application Development

Before working on this project, we had never developed an Android application before, so this was a challenge for us. We began by creating simple applications to display “Hello World” to help familiarise ourselves with how the application works.

We also read official documentation when learning about the features that we’d need to use, such as Toast and Fragments.

Separate from the official documentation, written and video tutorials were instrumental in helping us fully understand how to best implement some of the features of the application. StackOverflow was also a major help and was used frequently to bolster our grasp on the different topics we were researching.

Login System

When deciding on how to implement the login system, we had to consider which system best suited our needs. Early on, we made use of an SQLite database but upon further research, we found Firebase to be the best option as it provided us with a lot more options (cloud storage being one) and was simple to integrate into our application using Android Studio.

Design

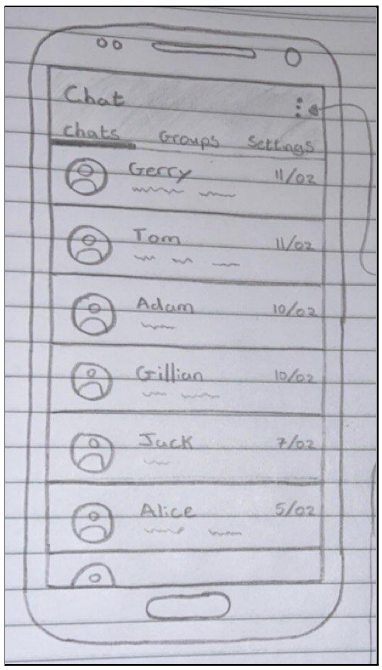
Designing the Frontend

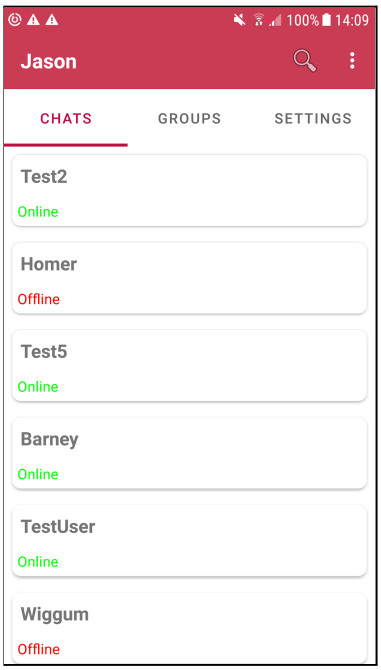
The frontend was designed in Android Studio. Android Studio allows users to create activities. Activities are single screens that consist of a Java class and an XML file. The XML file dictates the appearance of the screen and the Java class is typically in charge of the backend, such as deciding what happens when a button is clicked.

An example of this is the LogIn Activity. Its XML file is entitled *activity_login_layout.xml*. It contains two EditText fields (email address and password) and two Buttons (Log In and Sign Up). The Java file for this activity is entitled *LogInActivity.java* and its purpose is to control what happens when these buttons are clicked. For example, when the login button is clicked, it checks for any issues with the email address and password text fields. If there aren’t any, clicking the button will bring the user to the MainActivity. Otherwise, there is an issue with the email address or password entered and it will display the problem to the user.

When designing these activities, we also had to implement some services such as Firebase.

To assist us with designing the frontend, we both created sketches on paper to help us visualise how we wanted *Sentiscape* to look. Here is an example of a sketch opposite its implemented counterpart:

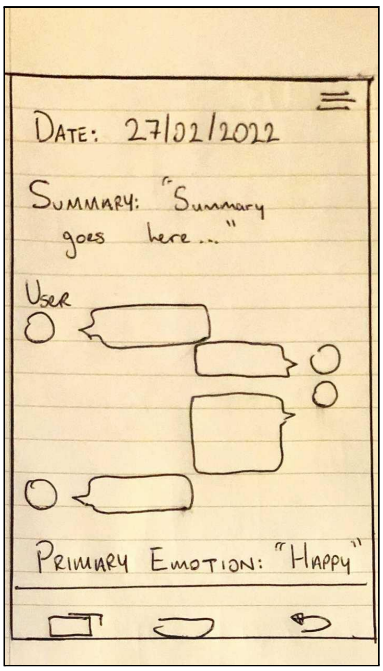


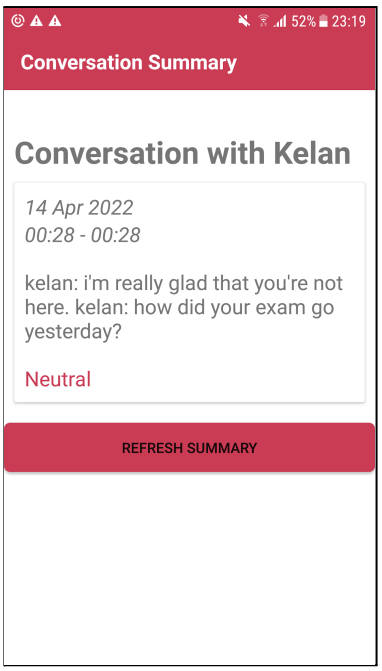


As seen above, having a sketch to base our design on was a large help and we were mostly able to match the drawn design in our implementation.

Sometimes our implementation would turn out different to our sketches, but they still provided valuable insight and forethought into how we wanted the application to look.

For example, the Summary Activity turned out differently than how we anticipated it to look:





We kept the date, summary and primary emotion. We then added the other user's name and removed the conversation display as it didn't add much to the screen and took up too much space.

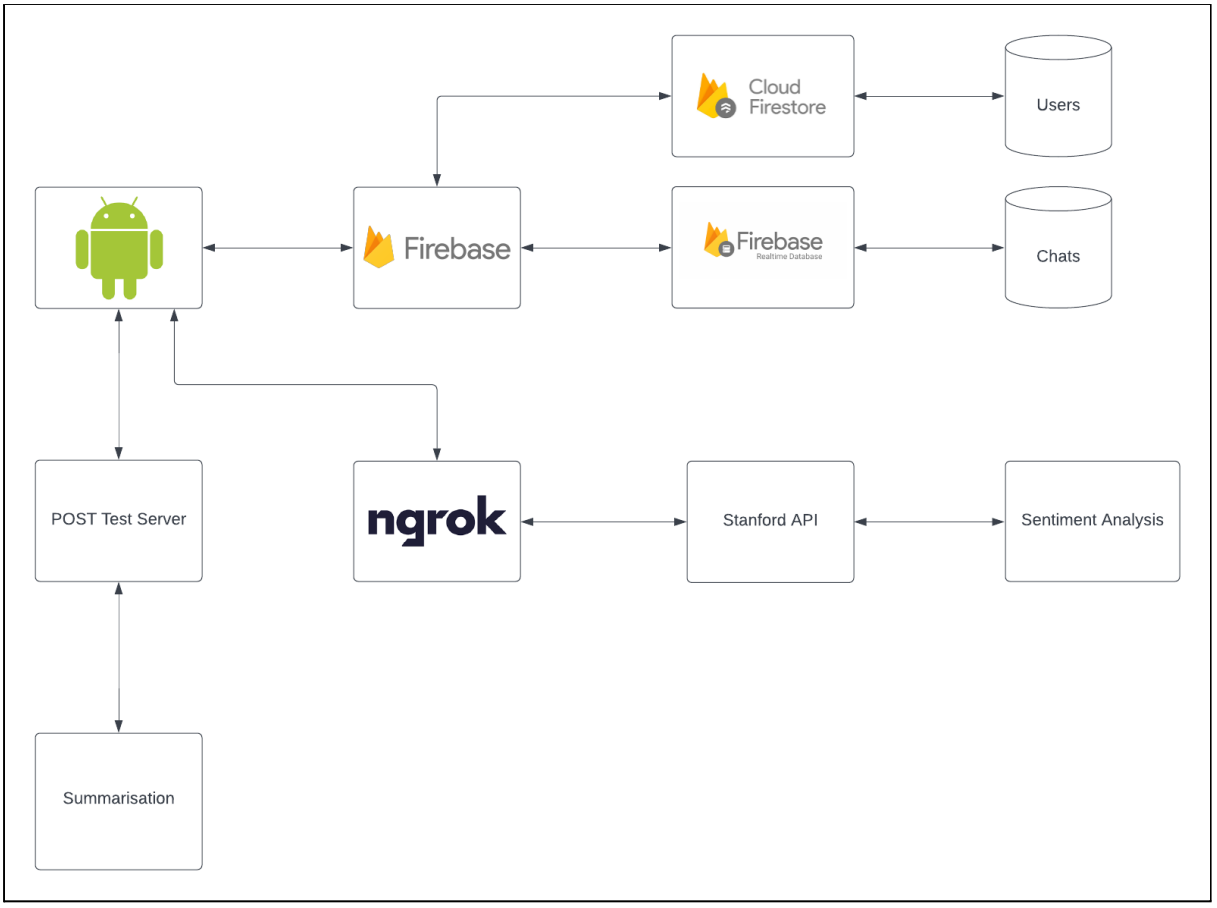
We would show these to our supervisor as a visual aid when discussing our next steps and we found that they helped a lot as our supervisor could more easily understand what we were trying to achieve.

Designing the Logo

The *Sentiscape* logo was designed as an SVG. This let us scale the image without worrying about the quality of the image deteriorating. When designing the logo, we chose to display the name of the application accompanied by a speech bubble containing a heart. This was sufficient enough at putting across the idea behind *Sentiscape*.

Design Diagrams

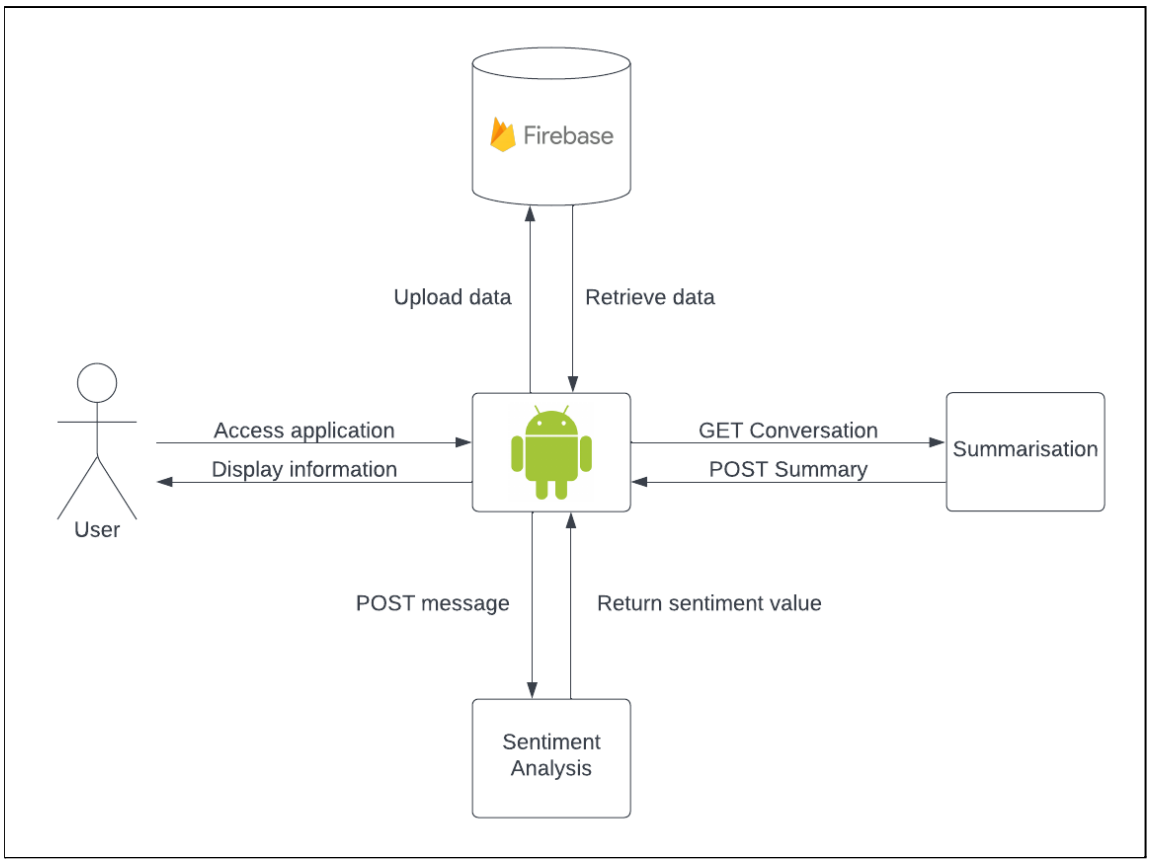
System Architecture



As seen in the above system architecture, the Android application is the centrepiece and is what the user will interact with. The backend of the application makes a series of calls to various APIs and databases in order to return the required data.

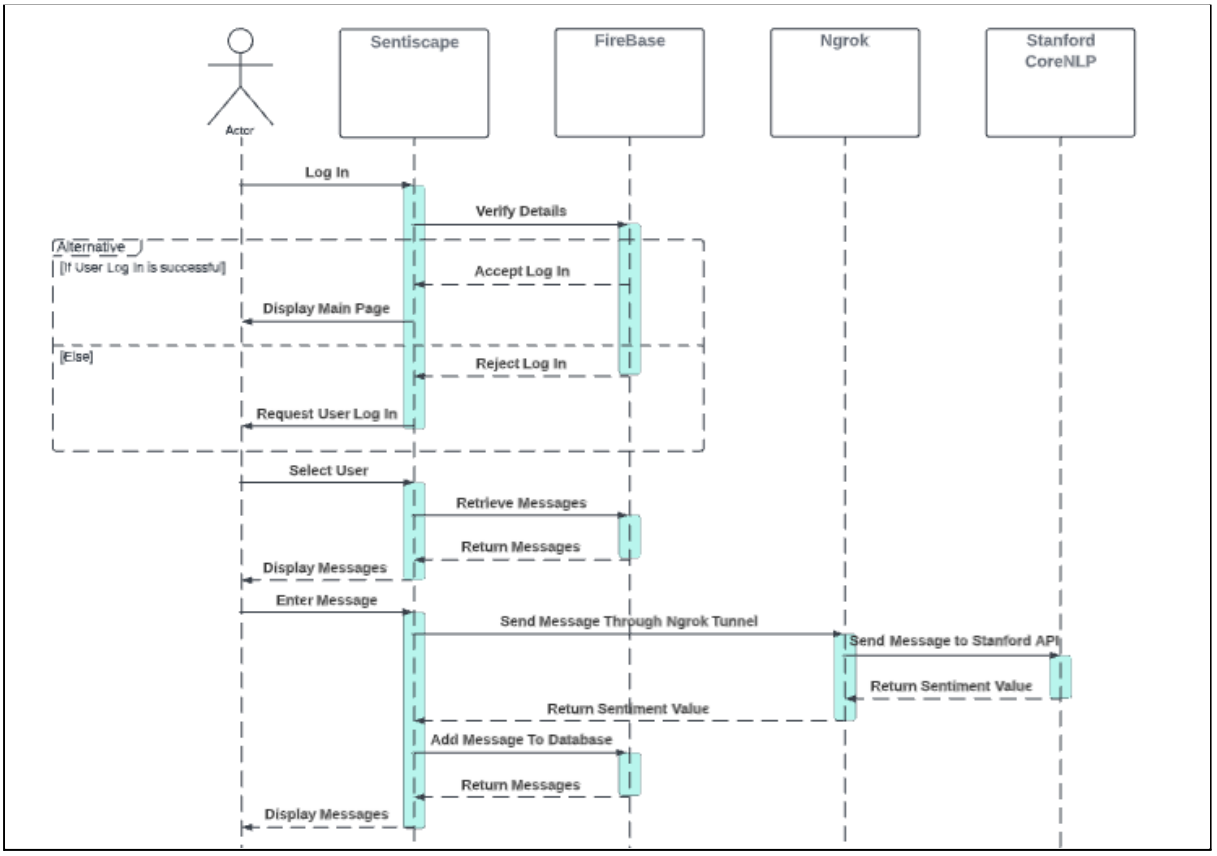
The application uploads data to Firebase's real-time database and the Cloud Firestore and retrieves the data when needed. Similarly, the application sends and receives POST and GET requests in order to calculate the summarisations. For the sentiment analysis portion of the application, there is a dedicated Stanford CoreNLP server being hosted on one of our local machines along with a Ngrok tunnel that allows external devices to send POST requests to the server and retrieve the sentiment value of a given text.

Context Diagram

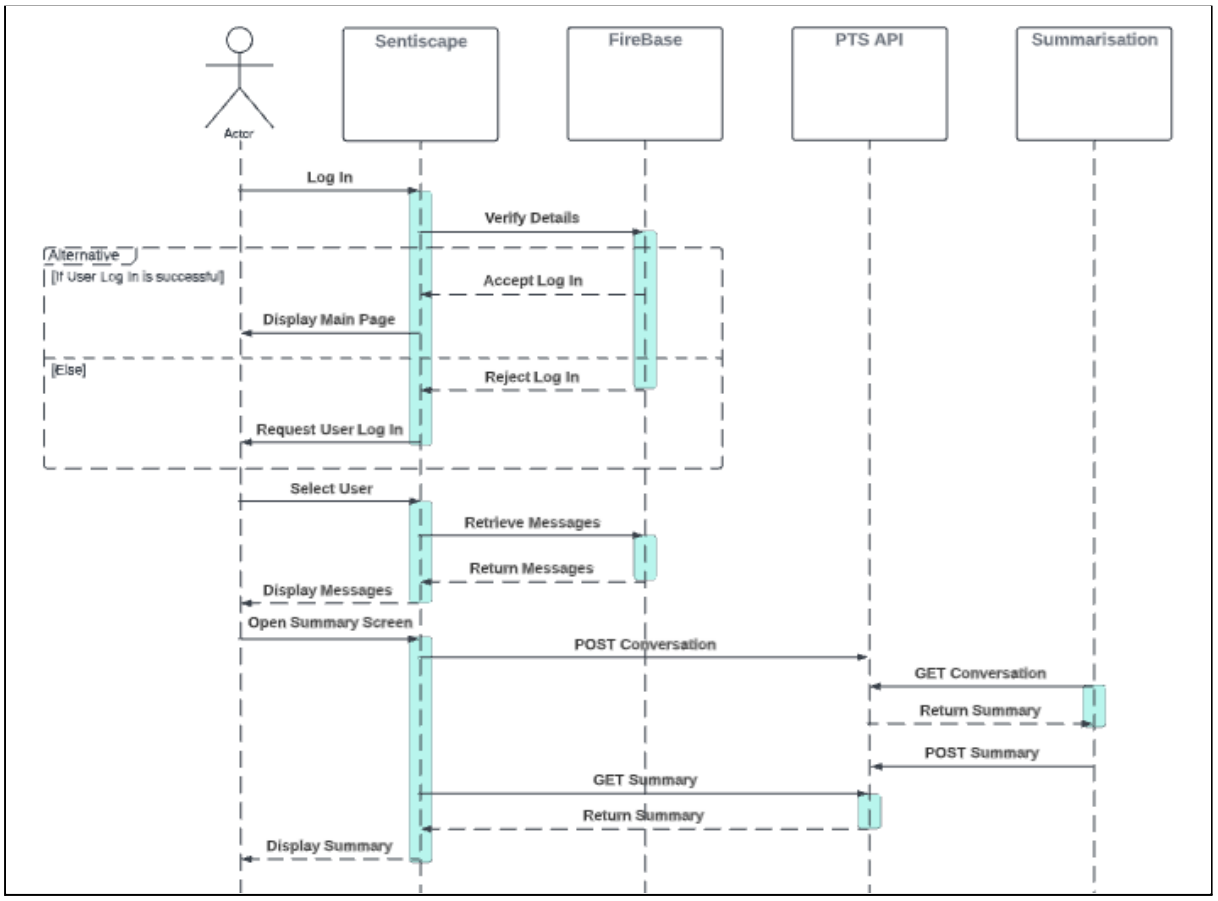


Sequence Diagrams

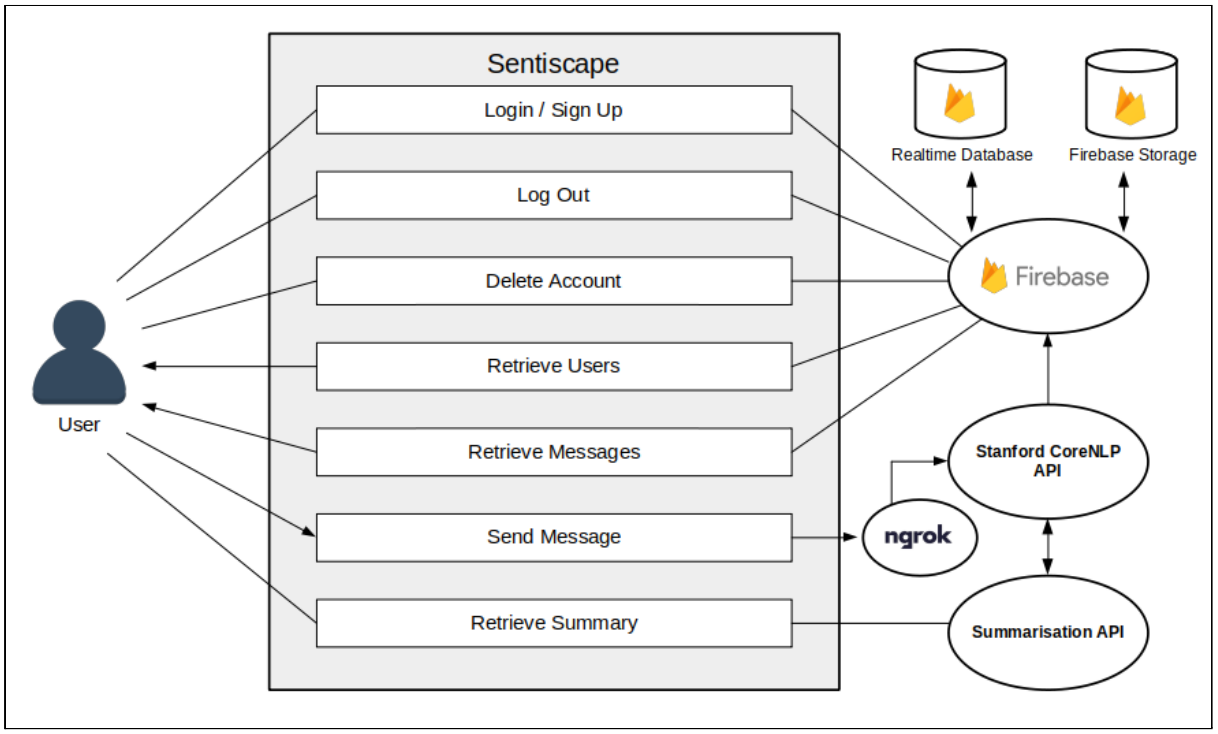
User sends a message:



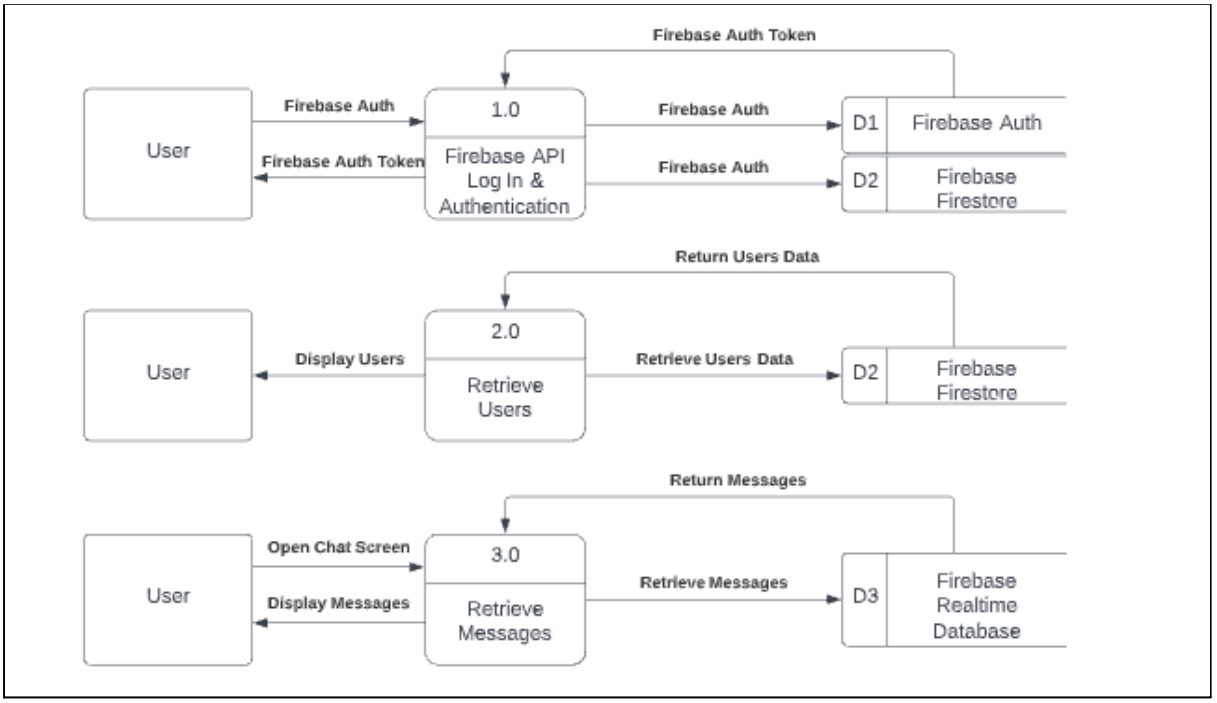
User retrieves summary:

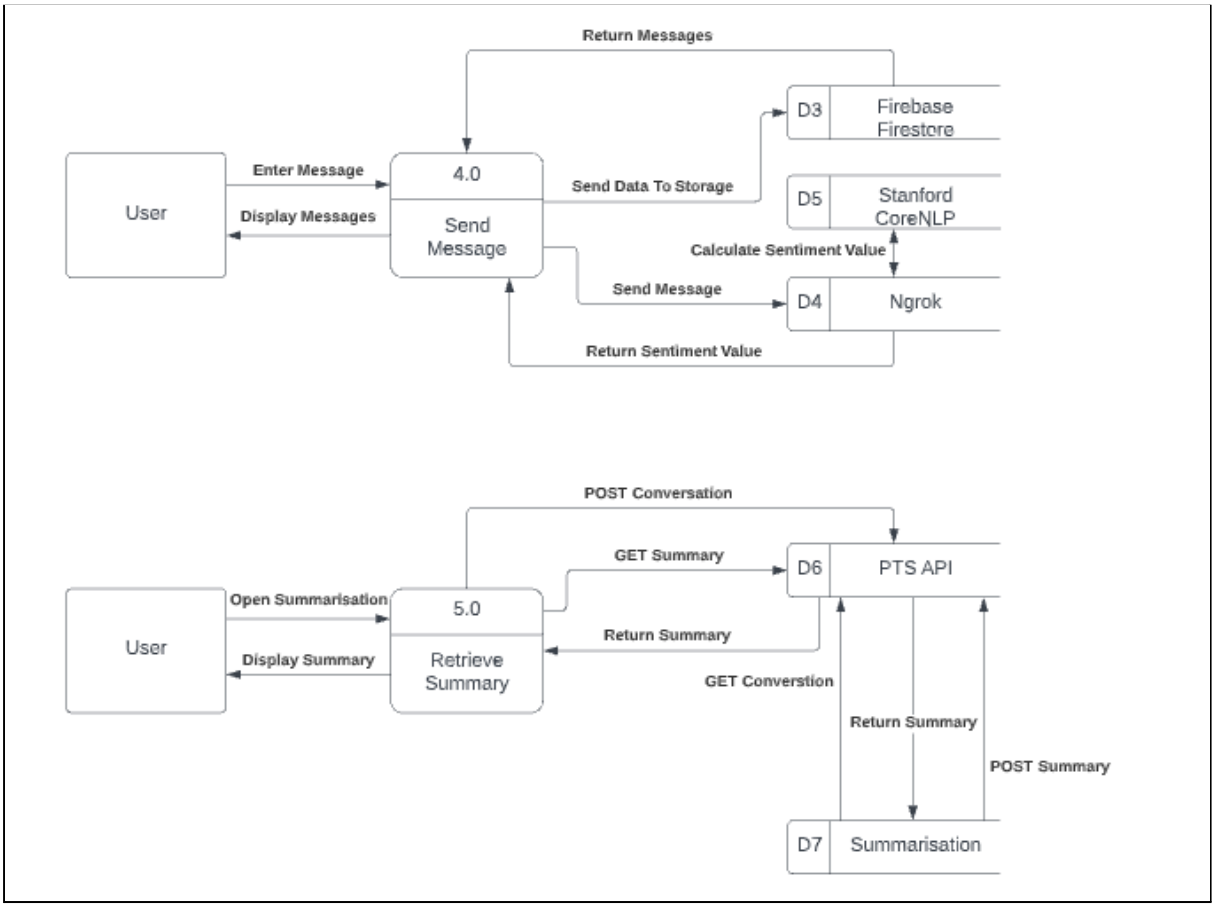


Use Case Diagram



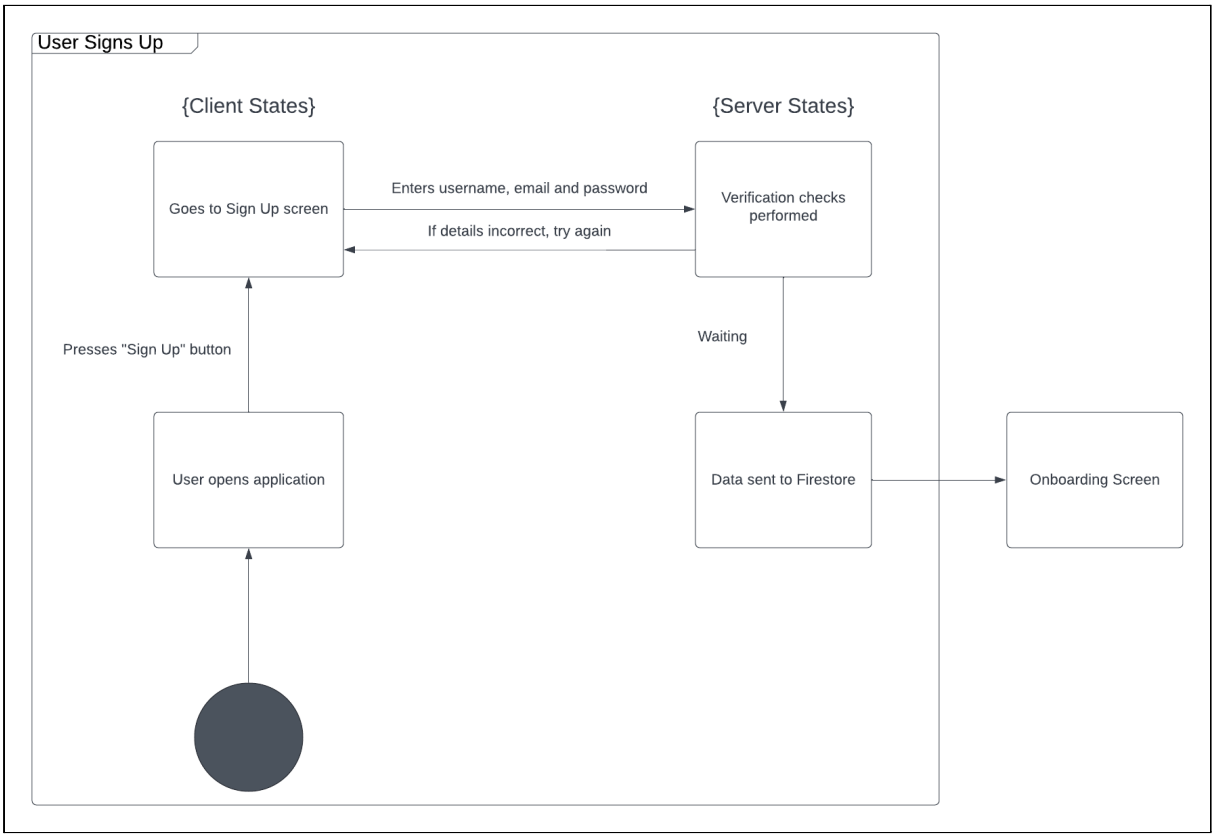
Data Flow Diagrams



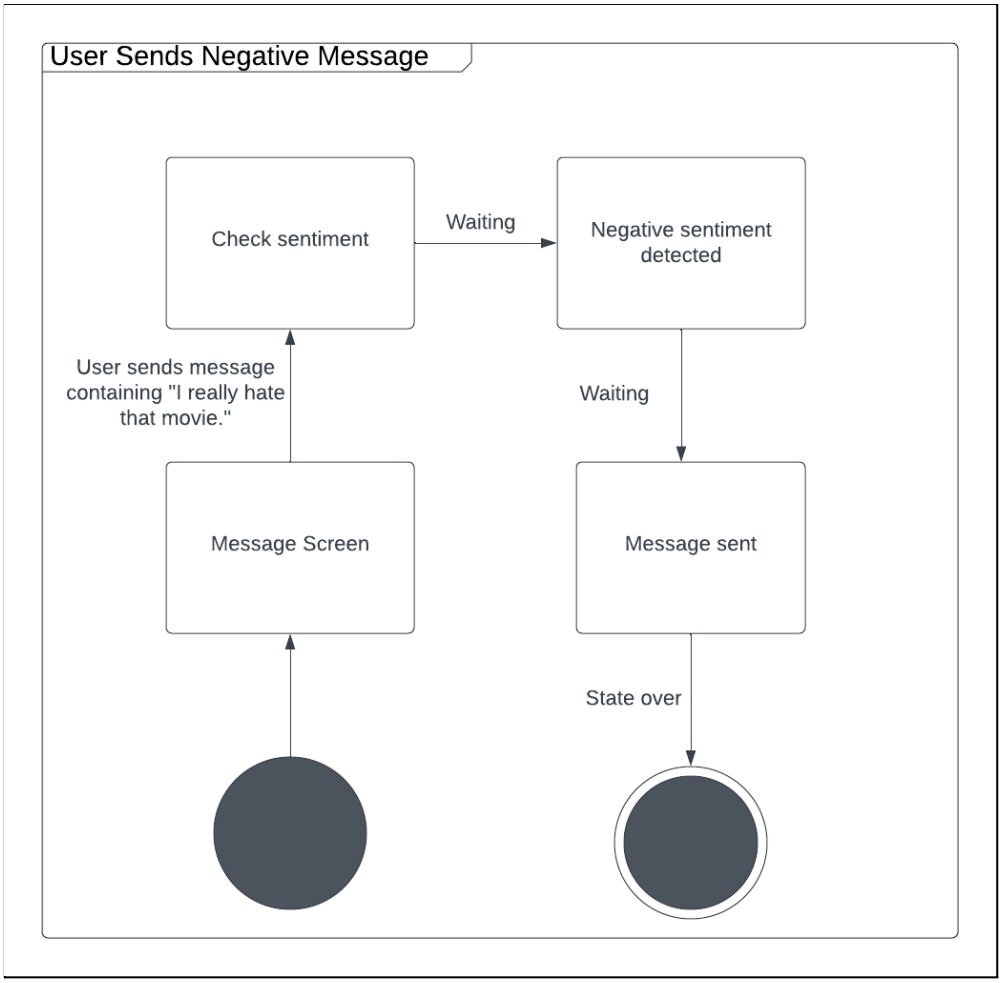


State Machine Diagrams

User Signs Up



User Sends Negative Message



Implementation

Lots of different features were implemented and integrated together to create *Sentiscape*. Descriptions of the implementation alongside sample code are included below.

How each language was implemented

Java

Java was used to implement the backend of the application. Android Studio was the IDE chosen and allowed us to create activities. Each activity represents a screen and usually contains a Java class and a layout file. The Java class handles what happens when buttons are pressed, switches are clicked, text is entered, etc. The following code snippet shows a Java function for a button that can be pressed:

```java
signUpButton.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        Intent intent = new Intent(getApplicationContext(), SetProfileActivity.class);
        startActivity(intent);
    }
});
```

XML

XML files were generated when new activities were created. These layout files were used to create the frontend of the application. Here we were able to style the different buttons, switches, headers, etc. and create a consistent look and feel across the entire application. The margins between different elements of the page can also be changed. Here is an example of a button designed in XML:

```xml
<android.widget.Button
    android:id="@+id/signUp"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_below="@+id/logIn"
    android:layout_marginTop="30dp"
    android:background="@drawable/button_background"
    android:text="Sign Up"/>
```

Python

Python was used to implement the Summarisation feature. It made use of multiple imported libraries such as Transformers, Tokenizers, PyTorch and Sentence Piece. The Summarisation feature was implemented using these libraries and a POST request is sent to a server to be retrieved by the application. Here's a snippet of the program used to summarise the conversation in Python. The Tokenizers library can be seen on the first line:

```python
# encode the text into tensor of integers using the appropriate tokenizer
inputs = tokenizer.encode("summarize: " + test1, return_tensors="pt", max_length=512, truncation=True)
# generate the summarization output
outputs = model.generate(
    inputs,
    max_length=150,
    min_length=16,
    length_penalty=2.0,
    num_beams=4,
    early_stopping=True)
```

Splash Screen Activity

The splash screen was implemented in a simple manner. The XML file contains an Image View (the *Sentiscape* logo) and the Java class displays the screen for 3 seconds before moving on to the next screen, the Login screen.

```
//  Splash screen displays for 3 seconds.
    private static int SplashTimer = 3000;
```

```
public void finishTiming() {
    Handler h = new Handler(Looper.getMainLooper());
    h.postDelayed(new Runnable() {
        @Override
        public void run() {
            //              App then transitions to login activity.
            Intent intent = new Intent( packageContext: SplashScreenActivity.this, LogInActivity.class);
            startActivity(intent);
            finish();
        }
    }, SplashTimer);
}
```

Login System

The login system was implemented using Firebase. It consists of the Login Activity and the Set Profile Activity. When the splash screen moves to the login screen, the application checks if the user is already logged in using Firebase Authentication, and if so, it skips the login screen entirely and moves to the Main Activity:

```
firebaseAuth = FirebaseAuth.getInstance();

if(firebaseAuth.getCurrentUser() != null) {
    Intent intent = new Intent(getApplicationContext(), MainActivity.class);
    startActivity(intent);
}
```

Otherwise, it continues to display the user the login screen where they can either log in using existing credentials or press the sign up button to advance to the Set Profile Activity.

There are four different services available to the user within the login system: logging in, signing up, logging out and deleting an account. They are further detailed below.

Logging In

If logging in, the user can enter their email address and password. The system will then perform various checks to ensure that the login details are correct, for example, to check that neither field was left blank:

```
if(sEmail.isEmpty() || sPassword.isEmpty())
    Toast.makeText( context: LogInActivity.this,  text: "Username or password not entered.", Toast.LENGTH_SHORT).show();
```

The user can then click the “Log In” button which leads to the Main Activity.

Signing Up

If the “Sign Up” button is pressed, the application advances to the Set Profile Activity. This activity uses Firebase Authentication to create an account consisting of an email address, username and password. Once the “Sign Up” button is pressed, the system performs more detailed checks than the Login Activity in order to create an account. For example, passwords must be at least six characters long:

```
else if(sPassword.length() < 6)
    Toast.makeText( context: SetProfileActivity.this,  text: "Password must be greater than 6 characters.", Toast.LENGTH_SHORT).show();
```

If all these checks pass, the account is created and the data is sent to Cloud Firestore. After this, the application advances to the Onboarding Activity.

Logging Out

The “Log Out” button is on the Settings Fragment. This uses the `getInstance().signOut()` function from Firebase Authentication. It then brings the user to the Login Activity.

```
FirebaseAuth.getInstance().signOut();
Intent intent1 = new Intent(getApplicationContext(), LogInActivity.class);
startActivity(intent1);
finish();
```

Deleting an Account

The “Delete Account” button is on the Settings Fragment. This removes the user from the real-time database on Firebase and from the Firestore. After deleting the data, the user is brought back to the Login Activity. They won’t be able to log in using their old credentials and will have to make a new account in order to sign in.

```
  Deleting account from realtime database.
user.delete()
```

```
  Deleting account from Firestore.
firebaseFirestore.collection( collectionPath: "Users").document(firebaseAuth.getCurrentUser().getUid()).delete();
```

Onboarding Activity

After signing up for the first time, the “Sign Up” button leads to the Onboarding Activity, which briefly explains how the application works. This was implemented using PaperOnboarding, a material design UI slider, created by Ramotion.

Each screen in the onboarding activity is created using five parameters: a title, a description, a background colour, an image and a page counter. Once all the screens have been created, they are stored in an array list. Fragment Transaction is used to perform all the transactions between screens.

On the last screen, users must swipe once more to exit Onboarding and enter the Main Activity. This is accomplished by using PaperOnboarding’s `onRightOut()` function:

```
paperOnboardingFragment.setOnRightOutListener(new PaperOnboardingOnRightOutListener() {
    @Override
    public void onRightOut() {
        Intent intent = new Intent(getApplicationContext(), MainActivity.class);
        startActivity(intent);
    }
});
```

Main Activity

The Main Activity makes up the central hub of the application. It contains the Chats fragment and the Settings fragment.

The fragments are set up using a PagerAdapter. The PagerAdapter contains switch statements, each of which returns a different fragment:

```
public Fragment getItem(int position) {
    switch (position)
    {
        case 0:
            return new ChatFragment();

        case 1:
            return new GroupFragment();

        case 2:
            return new SettingsFragment();

        default:
            return null;
    }
```

In the Main Activity, a tab layout is created and for each position in the tab, the pager adapter calls a particular fragment and the fragment is displayed on the screen.

```
pagerAdapter = new PagerAdapter(getSupportFragmentManager(), tabLayout.getTabCount());
viewPager.setAdapter(pagerAdapter);

tabLayout.setOnTabSelectedListener(new TabLayout.OnTabSelectedListener() {
    @Override
    public void onTabSelected(TabLayout.Tab tab) {
        viewPager.setCurrentItem(tab.getPosition());

        if(tab.getPosition() == 0 || tab.getPosition() == 1 || tab.getPosition() == 2)
        {
            pagerAdapter.notifyDataSetChanged();
        }
    }
```

Chats Fragment

The Chat Fragment displays a list of users as well as their online status inside of a CardView. The CardView looks like this:

To make this display once for each user, a RecyclerView is implemented. RecyclerView simply recycles the card view as many times as needed. The code for the RecyclerView can be seen below:

```
recyclerView.setAdapter(chatAdapter);
```

```
chatAdapter = new FirestoreRecyclerAdapter<FirebaseModel, NoteViewHolder>(allUsernames) {
    @Override
    protected void onBindViewHolder(@NonNull NoteViewHolder holder, int position, @NonNull FirebaseModel model) {

        holder.particularUser.setText(model.getName());
        if (model.getStatus().equals("Online")) {
            holder.userStatus.setText(model.getStatus());
            holder.userStatus.setTextColor(Color.GREEN);
        } else {
            holder.userStatus.setText(model.getStatus());
            holder.userStatus.setTextColor(Color.RED);
        }
```

As seen above, `recyclerView` sets the adapter to `chatAdapter`. For each CardView in the recyclerView, `chatAdapter` gets the username and online status from Firebase (and depending on if online or offline it will change the colour of the text).

Interlinked with this RecyclerView is a search bar that will filter users. This makes it easy to find users to text.

If a user's card is clicked on, the application will begin the SpecificChatActivity, where two users can message each other.

Settings Fragment

The Settings Fragment displays some features that allow users to configure their experience. Here they can click on the "Log Out" button in order to log out. They can press the "Delete Account" button to delete their account.

Specific Chat Activity

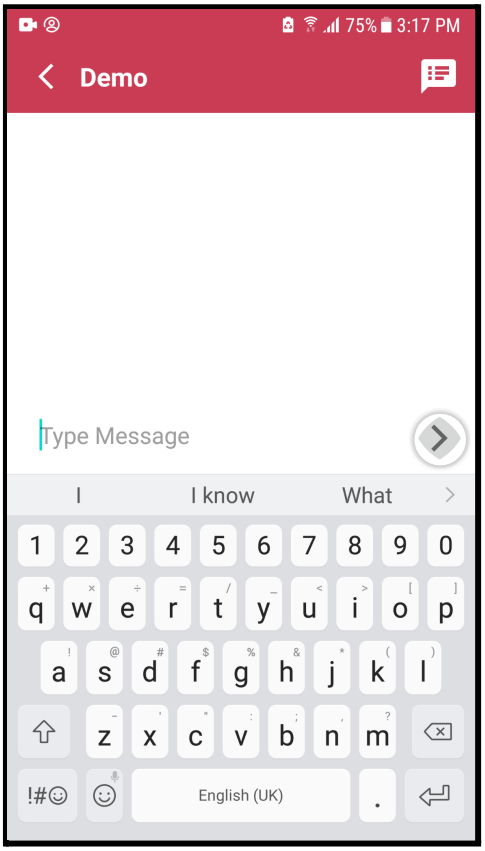
To access the Specific Chat Activity the user must select a candidate to message in the chat fragment by selecting one of the available users. The chat activity has a simple UI that allows the user to enter messages and scroll through previous messages while also giving access to the Summary Activity via the toolbar.

Once the activity is opened the application automatically retrieves any messages sent between the user and their chosen recipient by accessing the Firebase Realtime Database and returning the relevant messages. To retrieve the relevant information from the Realtime Database each conversation between users is saved in a "sender" and "receiver" chatroom to allow messages to be displayed correctly in the application, these chatrooms have a naming convention of concatenating the two users' UIDs to allow the application to fetch the data specific to the two users.

```
senderUID = firebaseAuth.getUid();
receiverUID = getIntent().getStringExtra( name: "receiverUid");
receiverUsername = getIntent().getStringExtra( name: "name");

senderRoom = senderUID + receiverUID;
receiverRoom = receiverUID + senderUID;
```

Sending messages between users is a simple operation for anyone operating the application as it follows the same conventions as traditional messaging applications, it has a text box for users to enter a message and a send button placed alongside the input text box.



When a user sends a message by hitting the send button a number of operations are triggered by the application, firstly a POST request is sent to the Ngrok tunnel which allows exterior devices to contact the Stanford CoreNLP server that is being hosted on one of our local machines. Once a POST request reaches the server containing the message that the user sent, the Stanford server calculates the sentiment value of the message returning a result between 1 and 4 (1 = Negative, 2 = Neutral, 3 = Positive, 4 = Very Positive).

```
JSONObject responseObj = new JSONObject(responseStr[0]);
JSONArray sentences = responseObj.getJSONArray( name: "sentences");
JSONObject sentimentObj = sentences.getJSONObject( index: 0);
Integer sentimentValue = Integer.valueOf(sentimentObj.getString( name: "sentimentValue"));
setSentimentValue(sentimentValue);
```

Set Sentiment Value

Once a sentiment value has been returned to the application the system moves onto the next stage of sending a message which is uploading the data onto the Realtime Database on Firebase. The message is uploaded to the “sender room” and “receiver room” along with the UID of the sender, the current time and date, the sentiment value of the message, and a timestamp.

```
Date date = new Date();
currentTime = simpleDateFormat.format(calendar.getTime());
Messages messages = new Messages(enteredMessage, firebaseAuth.getUid(), date.getTime(), currentTime, sentimentValue);
firebaseDatabase = FirebaseDatabase.getInstance();
firebaseDatabase.getReference().child("chats").child(senderRoom).child("messages").push()
        .setValue(messages).addOnCompleteListener(new OnCompleteListener<Void>() {
    @Override
    public void onComplete(@NonNull Task<Void> task) {
        firebaseDatabase.getReference().child("chats").child(receiverRoom).child("messages").push()
                .setValue(messages).addOnCompleteListener(new OnCompleteListener<Void>() {
            @Override
            public void onComplete(@NonNull Task<Void> task) {
                Toast.makeText(getApplicationContext(), text: "Message sent", Toast.LENGTH_SHORT).show();
            }
        });
    }
});
```

Upload to Database

When the message has been successfully uploaded the UI is updated on both the receiver end and sender end to display the new message, in addition the new message is pushed to the bottom of the screen so the users can always see new messages with ease.

```
messageRecyclerView.smoothScrollToPosition(messageAdapter.getItemCount() - 1);
```

The entire conversation background was also reactive depending on the cumulative total of the sentiment detected by each message. This allowed users to see how positive or negative each conversation was in general with each individual user.

```
if (messageAdapter.getItemCount() > 0) {

    Integer messageNumber = messageAdapter.getItemCount();
    Integer totalSentiment = 5;
    while(messageNumber > 0) {
        messageNumber = messageNumber - 1;
        if(messagesArrayList.get(messageNumber).getSentimentValue() < 2) {
            totalSentiment = totalSentiment - 1;
        } else if (messagesArrayList.get(messageNumber).getSentimentValue() > 2) {
            totalSentiment = totalSentiment + 1;
        }
    }
    updateUI(totalSentiment);
```

Total Conversation Sentiment Value

```
public void updateUI(Integer sentimentValue) {
    new Handler(Looper.getMainLooper()).post(new Runnable() {
        @Override
        public void run() {
            if (sentimentValue < 0){
                relativeLayout.setBackgroundResource(R.color.highNegative);
            } else if (sentimentValue < 2) {
                relativeLayout.setBackgroundResource(R.color.midNegative);
            } else if (sentimentValue < 5) {
                relativeLayout.setBackgroundResource(R.color.lowNegative);
            } else if (sentimentValue > 10) {
                relativeLayout.setBackgroundResource(R.color.highPositive);
            } else if (sentimentValue > 8) {
                relativeLayout.setBackgroundResource(R.color.midPositive);
            } else if (sentimentValue > 5) {
                relativeLayout.setBackgroundResource(R.color.lowPositive);
            } else {
                relativeLayout.setBackgroundResource(R.color.neutral);
            }
        }
    });
}
```

Set Background Colour

Summary Activity

The Summary Activity can be accessed by clicking on the Summarisation button in the toolbar of the Specific Chat Activity. This activity displays the summary of a conversation between two users.

When the activity is loaded, a POST request containing JSON data is sent to the server by the POSTConversation class. The JSON file contains the start and end time of the conversation, the conversation, the average sentiment score and the unique chat ID.

```
JSONObject postData = new JSONObject();
try {
    postData.put( name: "conversation", conversation);
    postData.put( name: "sentimentValue", sentimentValue);
    postData.put( name: "chatUID", chatUID);
    postData.put( name: "startTime", startTime);
    postData.put( name: "endTime", endTime);
}
catch (JSONException e) {
    e.printStackTrace();
```

A GET request is sent to the server and acquires the latest JSON data. This JSON data contains the date, timespan, summary and sentiment of the conversation. Each value is set to a specific text field on the Summary Activity for the user to see. Here, the JSONObject "FormValues" contains all the relevant key, value pairs and the "date" value is extracted from the data. The function `updateDate()` is run with the date as an argument which sets the text field to display the date.

```
JSONObject jsonObject = new JSONObject(retSrc);
JSONObject tokenList = jsonObject.getJSONObject("FormValues");
JSONArray dateArray = tokenList.getJSONArray( name: "date");
String jsonDate = dateArray.toString();
Log.i( tag: "JSON DATE", jsonDate );
updateDate(jsonDate);
```

Depending on the sentiment displayed, the CardView will change the background colour.

```
if (substr.equals("Negative")) {
    cardView.setCardBackgroundColor(ContextCompat.getColor(SummaryActivity.this, R.color.negative));
} else if (substr.equals("Neutral")) {
    cardView.setCardBackgroundColor(ContextCompat.getColor(SummaryActivity.this, R.color.neutral));
} else {
    cardView.setCardBackgroundColor(ContextCompat.getColor(SummaryActivity.this, R.color.positive));
}
```

If the user feels the data is outdated, they can refresh the page by clicking on the "Refresh Summary" button. This button reloads the page, which in turn sends a GET request to the server and returns the most recent JSON data.

Summarisation.py

The summarisation feature was implemented using Python. The primary library used was Transformers by HuggingFace [7]. Transformers provides pre-trained models on which to test data which saves time and resources that could be spent on other aspects of the project.

To collect the information from the application, the method `server()` performs a GET request on the server. This GET request returns a JSON file which is filtered to display just the body. Here's an example of what the body looks like:


```
{"conversation":"Will this go to the top?. I didn't see that time. I know. What. I know. Well. Ah. Definitely. I know. What. Scroll. What. Sell your face. I know but like. What. Scroll please. Please. Damn. What. \nMessage. message. What. Message. What. What. I know. I","sentimentValue":"2.0","chatUID":"IQWYaCp0G2MIyOjhFR7XtLTIci03xbEhkNLeeRQ2JMBAURqtruoWMKG2","startTime":"16:07","endTime":"17:09"}
```


The Python script can then access the individual key, value pairs. Here's the code for this:

```
getter = requests.get('http://ptsv2.com/t/sentiscape_conversation/d/latest/json').text
json_data = json.loads(getter)
d = json.loads(json_data['Body'])
conversation = d['conversation']
sentiment_value = d['sentimentValue']
chat_uid = d['chatUID']
start_time = d['startTime']
end_time = d['endTime']
```

Each value has a different operation performed on it before it gets posted back. The conversation gets summarised using the `get_summary(conversation)` method. If the conversation is below a certain length, no summarisation is performed as it is already deemed short enough and the conversation is returned. Else, summarisation is performed on the conversation.

As mentioned above, the summarisation is performed using the Transformers library on a pre-trained model. The model used was trained on the SAMSUM corpus. An output is then generated in the form of abstractive summarisation.

This is performed using a pipeline, an object from the Transformers library that abstracts the most complex code and allows for several different operations to be performed on a model, such as the “summarization” operation seen below.

```
summariser = pipeline("summarization", model="linydub/bart-large-samsum")
```

The value `sentiment_value` is a float number that is the average of all the sentiment analysis scores. Running the method `sentiment_analysis(sentiment_value)` rounds the value to the nearest integer and then returns the corresponding sentiment.

```
if sentiment_value == 1:
    return "Negative"
elif sentiment_value == 2:
    return "Neutral"
elif sentiment_value == 3:
    return "Positive"
elif sentiment_value == 4:
    return "Very Positive"
```

The value `chat_uid` is the unique ID of the chat between two users. When sending a POST request, this value is included in the server URL so that the SummaryActivity displays the correct summary for two users.

```
r = requests.post(f'http://ptsv2.com/t/sentiscape_summary{chat_uid}/post', data = data)
```

The values `start_time` and `end_time` represent the time of the first message sent and the last message sent. They get concatenated into a string before the POST request is sent and is then displayed on the Summary Activity.

```
times = start_time + " - " + end_time
```

12 Apr 2022
16:07 - 18:20

The current date also gets sent to the Summary Activity as seen above using the `get_datetime()` method.

```
def get_datetime():
    dt = datetime.today().strftime('%d %b %Y')
    return dt
```

Once all of these values are added to a JSON array, they're then sent to the server using a POST request and the application retrieves it.

Sentiment Analysis

Sentiment Analysis is used throughout the application and can be seen in a number of different locations as it is used in the Specific Chat Activity, Summary Activity, and the Sentiment Analysis Activity.

As mentioned previously in the Specific Chat Activity the sentiment analysis operation is completed by the dedicated Stanford CoreNLP server hosted on a local machine and allows users access via a POST request through the Ngrok tunnel also hosted on the same local machine.

The Sentiment Analysis activity is an additional feature added to the application which can be accessed via the Settings Fragment. The activity has the sole purpose of allowing users to test out the sentiment analysis function used throughout the application without having to send a message to another user. The activity demonstrates the reactive change each detected sentiment makes to the UI of the application.

The most prominent section where the sentiment analysis is put to use is the Specific Chat Activity where it is used to detect the sentiment of each message sent and received by the user.

Problems Solved

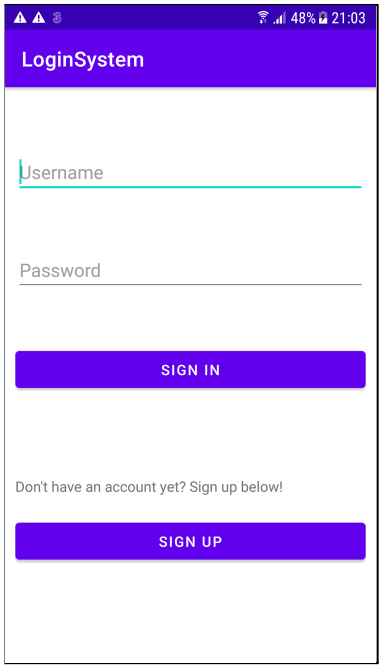
Over the course of the project, we ran into lots of problems thanks to mistakes made as well as our inexperience with certain tools and methods.

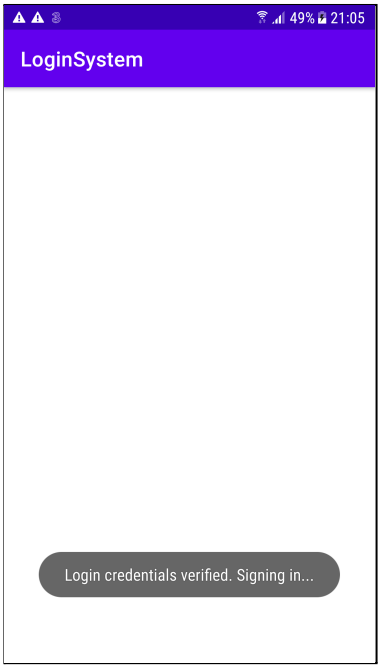
Deciding on a Login System

The login system was one of the first systems we worked on when developing the application. It was necessary to have an account system designed early on as the rest of the application depended on it. Unique accounts were needed so different users can message one another.

Due to our inexperience with account creation, we were uncertain of the best approach to implement this system. To solve this problem, we both opted to research login systems and design a simple login system to compare with each other.

After a day, we returned with two approaches. Approach A involved using an SQLite database to store a username and hashed password. The text entered for the username and password would be compared to what was in the database and if they matched then the user could log in.

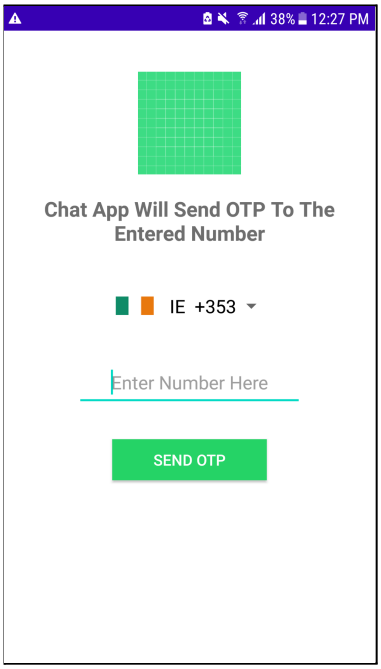


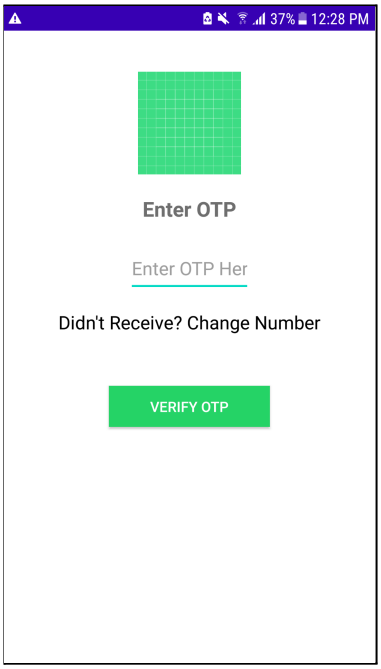


```java
public Boolean insertData(String username, String password) {
    SQLiteDatabase db = this.getWritableDatabase();
    ContentValues contentValues = new ContentValues();
    contentValues.put("username", username);
    contentValues.put("password", password);
    long result = db.insert("users", null, contentValues);

    if(result == -1) {
        return false;
    }
    else {
        return true;
    }
```

Approach B involved using Firebase Authentication to authenticate users. Users were verified using One-Time Passwords (OTPs). The user was to enter their phone number and an OTP would be sent to them via SMS. Users would then enter this OTP to log in.






```java
msendOtp.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        String number;
        number = mgetPhoneNumber.getText().toString();

        if(number.isEmpty())
        {
            Toast.makeText(getApplicationContext(), "Please Enter Number", Toast.LENGTH_SHORT).show();
        }
        if(number.length() < 10)
        {
            Toast.makeText(getApplicationContext(), "Please Enter Correct Number", Toast.LENGTH_SHORT).show();
        }
```

In the end, after discussions amongst ourselves as well as our supervisor, we decided a combination of the two approaches was most appropriate. We determined that Approach A's username and password system was a better fit than Approach B's OTP. On the other hand, we felt that Firebase Authentication was a better database to use than SQLite as the SQLite database is stored on a local device whereas Firebase is on the cloud. It also updates in real-time so we felt that this was the right approach.

Integrating Python with Java

The initial plan for the project was to design the application in Android Studio using Java. The machine learning aspect would be implemented in Python and then both would be integrated together at the end to create a fully functioning messaging application with sentimentality features.

At first, the machine learning Python programs were developed in parallel with the Android Studio programs. Eventually, we hit a point where we felt that the summarisation and sentiment analysis models were sufficiently developed and we were ready to integrate them into the main application. This was more difficult than we initially thought.

For the summarisation, it was set up to read in the messages as one large string and would output the current date and summarised conversation. The initial idea was to call this program to run directly in Android Studio. To accomplish this, we used Chaquopy. Once we had configured our Gradle build, we attempted to run the application using Chaquopy to integrate the Python script. This is what it looked like:

```
python.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        if(!Python.isStarted()) {
            Python.start(new AndroidPlatform(getApplicationContext()));
        }

        Python py = Python.getInstance();
        PyObject pythonFile = py.getModule("summarisation-2");
        PyObject summary = pythonFile.callAttr("summary");

        Toast.makeText(SummaryActivity.this, summary.toString(), Toast.LENGTH_SHORT).show();
```

Unfortunately, this didn't work. When the summary method was called on the line:

```
PyObject summary = pythonFile.callAttr("summary");
```

The application always crashed. The combination of the application running as well as the Python script executing was too resource-intensive and this caused the application to stop responding.

The next approach was to POST the conversation to a server from Android Studio. The Python script would run daily and GET the conversation from a server. The script would then summarise the conversation and POST a JSON file containing the date and summary back to the server. The Android application would then GET this JSON file and display it to the user.

In order to do this, we set the *summarisation.py* file to run repeatedly when the application is running. This isn't a perfect solution, but as it is a prototype we felt it was good enough and we had to prioritise other aspects of the application.

Setting up Firebase

Having no prior experience using Firebase we had some issues when setting it up initially with our project. Working on the project on separate machines meant that we had to create a singular account that we could both access Firebase with as Android Studio allowed us to connect our project with the Firebase tools we planned to make use of once we had an account connected to both.

There was also the issue of version control in the beginning as we had manually witten each of the dependencies we wished to use in the Gradle build file of our project. However we quickly realised that using the Firebase interface on Android Studio allowed us to implement the most recent version of each of the SDKs we wanted to use.

Hardware Issues

For the duration of the project, we spent the majority of the development process creating the application using our personal laptops. When working in different locations, we tended to call regularly and share our screens in order to show our work to one another. An issue we had with this is that we weren't able to share the working application as we couldn't show the app running on our devices and our laptops weren't powerful enough to share the screen and run the virtual device at the same time.

To try and solve this issue, we found it best to meet in person and work side-by-side whenever possible.

Another hardware issue that stemmed from not being able to use the virtual device was the lack of hardware we had available to use. The most recent Android device we had available for the bulk of testing was the Samsung Galaxy S6. Working with older Android devices led to issues as they lacked the processing power of more modern Android devices and they did not support some of the features we intended to implement locally.

We worked around this issue by hosting dedicated servers that the application could post requests to and return the relevant data after it had been processed on the server. This allowed us to bypass the issue of a device's processing power and reduce the size of the

application. Normally the main downside of hosting dedicated servers that the application relies on is the constant need for an internet connection, however, our online chat application already required an internet connection to send and receive messages so this did not add any additional hindrances to our application.

COVID-19 Impact

Over the course of the project, the ongoing pandemic had an impact as both members of the team caught COVID-19 which inhibited us in our work. We found it useful to meet in person in order to see how the application works on the Android devices and losing a couple of weeks to isolation affected us as we felt we weren't as productive in an isolated environment.

Results

Upon completion of the project, we successfully developed a series of different components and integrated them together. The best way to measure our results was to compare them to our functional specification, which detailed all of the requirements we planned to incorporate into our application.

First of all, we were able to build a fully functional login system, consisting of a login system, a signup system, a logout system and an account deletion system. These requirements all featured in the functional specification and were achieved fully.

Next, we were able to implement a search system that allows users to filter the user database and search for other users that they want to message.

We also aimed to build a network that allows users to message one another instantly. This was fully implemented into the final application.

With regards to the sentiment analysis functionality, the application is able to detect if users have sent a positive, negative or neutral message. Depending on the mood of the message, the colour of the message bubble will change and the message will be displayed alongside a smiling, neutral or frowning emoticon, respectively. This is incorporated as described in the functional specification.

Users are able to view summaries of conversations for each day and the timespan in which these messages were sent. It also includes the overall sentiment of the conversation. This was implemented slightly differently from how it was described in the functional specification, but overall the desired result was achieved.

Some of the features that were planned to be implemented were cut from the application due to time constraints and in some cases, they just weren't feasible. These include the group chat functionality, the conversation sidebar, and saving past conversations. When deciding

which features to include and which were cut, prioritisation was given to the features that we felt were most essential to delivering the application set out in the proposal and functional specification.

We also brought about some features that weren't originally part of the application such as the onboarding screens. This was not a vital feature, but it assisted users and was an added bonus for users and provided useful functionality (alongside features like dark mode).

Testing was an aspect of development that could have received more attention. Ad hoc testing was conducted routinely throughout the development process and user testing was analysed thoroughly, but unit testing was an element that could have been developed further. This was due to the fact that the application wasn't fully functional until the week leading up to the deadline. If we were to begin working on this project again, we would consider incorporating test-driven development as a means to thoroughly test the application and would offer a valuable learning experience as it is a very different process for software development.

Overall, we are satisfied with the resulting application and we feel that it accurately reflects the many hours of research and development undertaken by both members of the team. On reflection, we made plenty of mistakes, but these mistakes offered invaluable insight and lessons which we will be able to take with us during our future careers. Finally, we came away feeling that the project served as a perfect capstone for our undergraduate degree as it required us to apply numerous different skills and qualities that we had learned over the course of the four years in Computer Applications.

Future Work

In the future, there are a lot of possibilities for expanding and improving on the existing application. Due to time constraints, we were unable to implement all of the features that we initially planned on including.

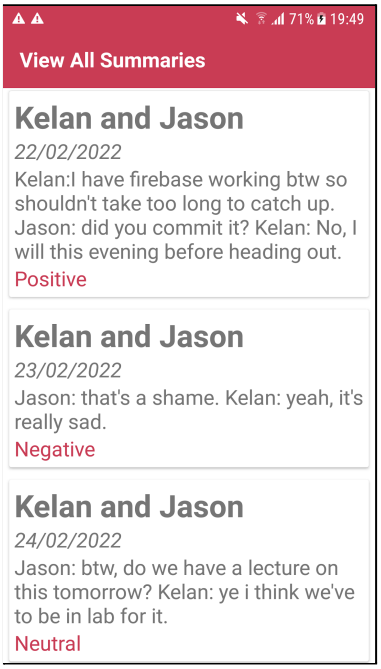
One feature that we didn't have time to implement was the group chats feature. As some aspects of the application took longer to develop than we had first thought, we realised that we would have to cut some features out. Of the features we had yet to implement, we discussed which was the least important and group chat functionality was the first to come to mind for both of us. It was a feature that would be a nice addition to the application but wasn't as necessary as some others. In the future, this would be a feature that we would like to look at again.

Another feature we were not able to implement was the conversation sidebar. We had some time, in the end, to try and implement this using Android Studio's NavigationUI class but we were unable to make it work as we wanted so instead of this we included a button in the action bar to display the Summarisation feature.

We would have also liked to have spent more time on training the datasets for our text summarisation and sentiment analysis models as there were times when they would be

inaccurate. However, the scope of the project was larger than originally expected. This meant we spent a lot of time working on the basic functionality of the application before getting to implement the machine learning features.

Although we tied together the Affection and Summarisation features in the Summary Activity, we were unable to extend this to include the initially included SummaryListActivity. The intention of this activity was to include a list of summaries from past days. An attempted implementation of this activity can be seen below:



In the future, we would like to re-work this activity and make it fully functional.

Finally, we had originally planned to let users include multimedia in their messages. This wasn't a part of the finished product in the end but could be included in future work. This could be expanded upon by including image classification to detect sentiment in images.

On top of this, in user testing, testers suggested a range of ideas that could be implemented, such as voice messages, profile pictures and notifications which would help flesh out the application from a prototype and bring it more in line with existing messaging applications such as WhatsApp and Facebook Messenger.

In conclusion, there are plenty of ideas that we could further investigate and develop but overall, we are satisfied with the application as it stands.

References

[1] A. Mertes, "Emotions and Inanimate Objects: ‘Kansei Engineering’ Explains How We Respond to Products", *Quality Logo Products*, 2021. [Online]. Available: https://www.qualitylogoproducts.com/blog/kansei-engineering-emotional-responses-to-products/. [Accessed: 18- Apr- 2022].

[2] Cherry, K., 2020. *How the Color Blue Impacts Moods, Feelings, and Behaviors*. [online] Verywell Mind. Available at: https://www.verywellmind.com/the-color-psychology-of-blue-2795815. [Accessed 26 March 2022].

[3] Cherry, K., 2021. *How Does Orange Influence Your Moods?*. [online] Verywell Mind. Available at: https://www.verywellmind.com/the-color-psychology-of-orange-2795818. [Accessed 26 March 2022].

[4] Cherry, K., 2020. *How the Color Blue Impacts Moods, Feelings, and Behaviors*. [online] Verywell Mind. Available at: https://www.verywellmind.com/the-color-psychology-of-blue-2795815. [Accessed 26 March 2022].

[5] J. Hartman, "Kotlin vs Java: What’s the Difference?", *Guru99*, 2022. [Online]. Available: https://www.guru99.com/kotlin-vs-java-difference.html. [Accessed: 09- Apr- 2022].

[6] Eric N. Forsyth and Craig H. Martell, "Lexical and Discourse Analysis of Online Chat Dialog," Proceedings of the First IEEE International Conference on Semantic Computing (ICSC 2007), pp. 19-26, September 2007.

[7] Bogdan Gliwa, Iwona Mochol, Maciej Biesek, and Aleksander Wawer. 2019. SAMSum Corpus: A Human-annotated Dialogue Dataset for Abstractive Summarization. In *Proceedings of the 2nd Workshop on New Frontiers in Summarization*, pages 70–79, Hong Kong, China. Association for Computational Linguistics.

[8] "Transformers", Huggingface.co, 2022. [Online]. Available: https://huggingface.co/docs/transformers/index. [Accessed: 12- Apr- 2022].

[9] “Stanford CoreNLP” [Online]. Available: https://stanfordnlp.github.io/CoreNLP.

[10] “Ngrok” [Online]. Available: https://ngrok.com.