

Aurora – Information Retrieval and Profiling

Report to the Examiner

Name: Jak Hall

Student Number: 15826673

Course: BSc Computer Science 3rd Year

Supervisor: Dr Gulden Uchyigit

Contents

Research and Planning.....	4
Background Research.....	4
Introduction	4
Historical Importance.....	5
Modern Approaches	5
Project Direction	9
Requirement Analysis	9
Diagrams	12
Entity Relationship Diagram.....	12
System Architecture Diagram	13
Project Lifecycle	13
Implementation	14
Phase 1 – Social Platform.....	14
Phase 2 - Data Analysis and Retrieval	24
Phase 3 – Mobile Application	36
Testing.....	46
Functional Testing.....	46
Performance Testing.....	47
User Testing	48
Summary	49
Evaluation	49
Reflection	49
Future Improvement.....	50
References	51
Appendix	54
Aurora – User Guide and Demonstration	54
Introduction	54
Login Screen	54
Registration	55
Home Screen.....	56
Manage Documents.....	57

Guide	57
Editing Documents	58
Manage Users	59
Manage Details	60
Viewing Other Users	61
Search Bar	62
Suggestions	62
Aurora – User Testing	63
Introduction	63
User Assessment	63
Consent	66
Aurora – Project Log	67
Introduction	67
Log Table	67

Research and Planning

Background Research

Introduction

Within the last decade there has been an increase in attention to accumulating and managing vast amounts of data by big companies. It is commonly referred to as the new oil. This is largely because “Without large quantities of big data it is hard to be truly competitive” ⁴. Fundamentally it fuels the commercial tech industry and with the ‘Internet of Things’ on the rise it will affect our lives on a greater scale than ever before.

In abstract terms; Data is the assigned value of any measurable quantitative or qualitative variable. Its simplest form is raw data. On its own, it tells nothing about a given system. This makes it hard to comprehend its tangible significance. It requires to be processed in a way it can be compared to other data points to form processed data or information, which can then be given context in order to make a reasonable assumption about a model. In most cases the more data there is in a set, the more accurate the assumption can be – this makes it incredibly important in trend recognition and future predictions.

Facebook is a perfect example of a thriving digital economy that is driven by the collection of data. According to statista.com the company gained 269 million active users with an average data size of 500mb of data per user, they would have collected 134.5 petabytes of data on new users alone. It’s a misconception that the company then sells that data on to other external sources. Besides unintentional leaks such as the recent discovery of Cambridge Analytica, Facebook claims all data stays internal.

How the company then generates revenue from that data is by extensive data analysis that aims to profile its users on measurable traits and preferences. External businesses can then request to advertise to a select group of users based on this profiling. It also uses the data for improving their own platform for a better user experience such as facial recognition on images. The intentions are to protect your personal data and use it safely and tentatively.

There is however a public fear in the vast collection of data. People don’t like personal details being in the hands of a centralized system that has full control over it – this raises the question of to whom the data belongs to and to what extent can companies utilize it. In such an unregulated field its likely governing restrictions will applied to companies to conform to the public expectation for the digital industry.

A recent example of this is the GDPR recently enforced by the European union to replace the original data protection act of 1995. Its primary focus is about the collection and storage of personal information. Under data subject rights an individual has must have access the information “whether or not personal data concerning them is being processed, where and for what purpose” ⁵. It’s clear a lot of companies will need to make big changes to comply with these new policies.

To understand the problem appropriately however it requires knowledge of how big data systems operate and what affect they have socially and economically. It's equally a public concern that people enforcing regulations don't have the experience or skill set to deal with these technical issues and if regulations continue, they will result in a restriction of technological growth.

As a result, I have decided to research how these systems function and implement one of my own in a way that reflects data retrieval and analysis as well as data transparency.

Historical Importance

Information retrieval has been around for centuries. An early example of centralized data collection on a large scale can be traced back to 1086 with the invent of the 'great survey' or colloquially dubbed the 'Domesday book' which was introduced by King William the Conqueror to analyse the land and social distribution of England and Wales. The first copy was one bound book written by likely a single scribe. It was a problematic system however served its purpose to quickly obtain information for the king's council.

In 1804, Joseph Marie Charles invented the 'Jacquard machine' as a device fitted to a power loom that could generate predefined patterns using punch cards. It's notable as it is the first demonstration of automated information retrieval for a specific task. Many early computers then followed a similar methodology for creating variable system processes based upon the user parameters.

With the invent of the internet there became a problem similar to 11th century England in that there was a lot of data with no way of congregating it in a usable fashion. It needed to be processed into comparable values. The first search engines used methods such as term weighting in order to automate this in a reliable way making vast amounts of data traversable in a short period of time.

Modern Approaches

Information Retrieval

Information retrieval (or IR for short) is a field of study that focuses on automated data extraction using a combination of mathematical algorithms and heuristics, for the "acquisition, organization, storage, retrieval, and distribution of information" ². If the method of IR is suitably efficient, the

obtained information will be considerably more valuable for numerous applications – such as pattern recognition and user profiling.

IR can be very helpful for the inclusion of user queries in a system. If there is a vast amount of documents or webpages accessible by a user they may require far less than 1% of the total data; this can be measured by the information entropy of a stochastic data source. If the entropic value is very low, the more complex and specified a system's query may need to be.

'Keywords' is the simplest and most common query in identifying relevant information within a text database, however more complex queries, such as 'Boolean' queries which concatenate a series of parameters together, 'Phrase' and 'Proximity' queries which seek out strings of words in a strict or relaxed fashion respectively. As well as 'Full Document' queries which searches for similar documents as the user submitted document, and lastly 'Natural Language Questions' which is the most complex query example in which a users informational need is requested in the form of natural language. This last query is in active research and could be incredibly useful in the future of information retrieval.

A method of identifying keywords is 'Term Frequency – Inverse Document Frequency' or TF-IDF which takes the number of times a word appears in each document and divides it by the number of documents the word appears in. What this does is identifies defining terms – which are rare from the scope of the whole text database but abundant in a specific portion of text. A list can then be drawn for each document of words which best represent the document and are each given a score. So, although a word such as 'the' appears many times in the text, it will have a low score as it is so common within other text and is therefore indistinctive.

$$TF(t) = n(t)/n(d)$$

$$IDF(t) = \log(TF(t)/DF(t))$$

Where:

- **n(t)** is the number of times a specific word **t** appears in a document.
- **n(d)** is the total number of terms in the document.
- **TF(t)** is the term frequency of a word.
- **DF(t)** is the number of documents containing that word.
- **IDF(t)** is the calculated inverse document frequency.

The above equations are commonly used to calculate the TF-IDF, a logarithmic scale is often used as the results are often exponential creating an impractical score range. It's also important to recognise that TF-IDF works better at identifying key terms the more documents are present in the text database.

Vector Space Model

To compare the document in a system to others, each term TFIDF weighting can be thought of as a different axis on a mathematical graph, this forms a multidimensional vector object which a coordinate position can be assigned to. A vector space model (VSM) considers all documents In the

system and calculates the position of each one. How similar two documents are depends on spatial proximity.

Generally, to compare the proximity between two-point vectors the Euclidean distance would be calculated between them. However, in this case magnitude is negligible and therefore it is better measure the angle between them. This is done using cosine similarity which takes the dot product of the two vectors and divides it by the product of both vector normal. This will return an angle in degrees or radians, the arc cos of this value can then be used to find a value between the range of 0 and 1, 0 being entirely different and 1 being identical in lexicon used.

$$\text{Normal Vector} = \sqrt{W_1^2 + W_2^2 + W_3^2 \dots + W_n^2}$$

(Where W is the weight of each axis.)

$$\text{Dot Product} = V_1W_1 * V_2W_1 + V_1W_2 * V_2W_2 + V_1W_3 * V_2W_3 \dots + V_1W_n * V_2W_n$$

(Where V is the vector number.)

$$\text{Cosine Similarity} = \cos^{-1} \frac{\text{Dot Product}(v1, v2)}{\text{Normal}(v1) * \text{Normal}(v2)}$$

Calculating similarity in this way is very efficient for systems that rely on accurate and fast results. Additionally, vector objects can be made for things other than documents – queries can be calculated within the VSM to compare them to all documents for fast and complex searches. As well as profile preferences which can help to locate relevant content to the user and is a large part of recommender systems.

A standard VSM will be generally stored as a matrix; rows being each of the terms and documents being each column. This means terms that the document does not include will still have a position in the matrix and must be denoted as 0, as comparing every term the VSM is important for calculating the cosine similarity. This means traditionally the matrix will contain more void cells than weightings. To reduce data redundancy the sparse matrix can be stored in a format that eliminates all blank positions and can recalculate the original VSM when needed without storing an excess of unnecessary data.

For me, information retrieval will be imperative to my project, if I'm to create a system capable of storing a large amount of text documents; then the user must be able to obtain the appropriate information. They will hopefully be able to do this by entering complex queries that will produce accurate search results. Analysis of information entropy can also be used to identify data redundancy within my text database which could be excluded to improve efficiency.

I have also created a prototype java project which utilizes TF-IDF to identify key terms, I intend to add querying functionality to the prototype to test the accuracy of my system. If the results are positive then I will be including the algorithm into my main project.

User Profiling

User profiling is a useful tool to characterize a systems profiling model with a user's personal interests and information. There are many advantages to profiling; HCI design can be tailored to a users preferences and also reduces HCI clutter through information filtering. More importantly users will be able to find and access relevant documents and also have documents suggested to them based on data the system has collected about the user.

User profiling works by identifying patterns in user activity and user submitted information. If a user has a history of looking at specific data, data of similar characteristics will have a higher probability of relevance to that user. Likewise if a user enters their age into the system, that information can be used to create classification grouping. When lots of users of a similar age view a specific document, that document could be a priority for other users of that age.

A recommender systems is a type of information filtering which aims to predict a user's preference towards an entity within the systems database. It uses classification algorithms and pattern recognition to make user suggestions, then relies on feedback in order to make better suggestions in the future. Recommender systems have many applications such as shopping, entertainment and travel planning; it can make companies a lot of money and help users make better decisions whilst also saving time.

Feedback is crucial to the recommender system, feedback can come implicitly and explicitly. Indirectly obtaining information about the user is less intrusive and reduces dishonesty. Examples of this are previous queries, purchases and repeat views to specific documents/pages a user has made. However direct feedback will be more telling of a users preference to a result and is very helpful in quickly building an effective system.

I will be applying user profiling in my system, as each user will be able to create documents and view other user's documents, the main objective of profiling is to display and suggest pages of similar content as well as provide a custom HCI experience tailored towards the individuals' preferences. I will attempt to acquire user data as unobtrusively, ethically and precisely as possible. I will be implementing a recommender system into my project. As I am making an application to contain a large amount of text documents, it's main function will be to sufficiently recommend documents which will be of interest to each user in my system. It will also collect information on user's activity, such as the users search history, the documents it views and other users they 'follow'. As well as respond to direct feedback it receives through document and user ratings in order to improve its accuracy.

Machine Learning

Machine learning is the ability for a computer system to improve at a certain task over time with no manual alterations to how it functions. Many problems in computing have heuristic solutions which are not easily measurable or obtainable by any conventional algorithm. An intelligent system will aim to recognise patterns in data, gather statistical analysis over time and then use that to make its own estimation of a problem.

Feedback data will then score the system on the accuracy or efficiency of a solution – it will then take the information it has 'learnt' from the session and take it into consideration next time it is asked to solve a problem to ultimately improve its score rating. How quickly it is able to improve the average score rating is an indication of the effectiveness of the implemented machine learning.

Supervised learning is when the system is given information it knows to be correct, this is inputted into the system as the training set. The training set is comprised of both an input and output objects, along with it a test set is provided comprised of exclusively input objects, the system must then produce output objects for this test set which it is then rated upon. Unsupervised learning is less common and more difficult to produce, however very helpful for when no training data is available and is often used to recognise data patterns which are not manually noticeable.

Clustering is often used in data mining to classify data into different categories. It is a common exercise to measure the effectiveness of a system by getting it to label a series of objects based upon their characteristics, the optimal result is that it will label every data point correctly.

"Documents in the same cluster behave similarly with respect to relevance to information needs." 1

– This infers the solution to a problem the system provides may be based upon what categories an entity falls into.

Machine learning would be a valuable inclusion into my project, especially to improve my recommender system which will rely on user feedback, however the feedback will be useless if the system cannot making improvements to its own prediction over time. I will likely be practising supervised learning but providing training data to my application, I will calculate the score rating to ensure it is improving with time.

I also hope to utilize machine learning to allow users to enter complex queries, such as natural language to search for the documents they want, this can be done by using classifying algorithms to automatically categorize the text in the system.

Project Direction

In my project I will use the knowledge I have attained from research into information retrieval and recommender systems to implement a design that will allow users to make fast queries of all stored documents, as well as receive suggestions that are appropriate for the users document creation and view history. Additionally, I will consider the factors of social platforms to produce my own network of users whilst respecting their necessity for data transparency.

To do this, I have decided to implement a system that will demonstrate the power of term weighting for user profiling and complex queries. However, I needed a user case for the project to provide real world application. I also did a lot of research on how social platforms collect data so It seemed appropriate to build an application that reflect these two ideals.

I therefore decided to build a mobile application that will allow multiple clients to connect to a centralized server, share content with other users and receive suggestions from the system once it has calculated the document and profile vector weightings. It will function as a hybrid between standard social networks and a forum site which expects longer, more formal text data.

During my time at university I have studied a large diversity of different computing skills, my project plan attempts to include as many of these as I can. Such as mobile app design, data structures and algorithms, intelligent systems, object oriented design and java development in general which has been the primary programming language taught over the three years.

Requirement Analysis

The system requirements of my system outline what I must achieve to produce a working application and network. I have broken it into three sections for the three phases of my development I have discussed with my supervisor.

A - Social Platform

Functional Requirements:

AF1 - The platform must be connected to a database which stores all user data (excluding documents and images) and be retrievable at any time the system is running.

AF2 - It must have a secure login which allows user to access their personal profile, it must check if the username and password are correct and create a new session once successful.

AF3 - It must allow users to create new documents and store images, title and document text for these documents server-side.

AF4 - It must allow users to access other user profiles within the system and view their documents.

AF5 - They must be able to “friend” other users and consequently receive new posts made by the specified user. As well as “unfriend” them if required.

AF6 - The user must be able change their profile image and any personal details at any time; besides their username which is unique to each profile.

AF7 - New users must be able to create a new account by completing a registration form.

AF8 - It should allow multiple clients to be connected at once.

User Requirements:

AU1 - Users should be able to receive suggested content to them which is specific to each account, displaying similar documents by other users.

AU2 - They should be able to search the database to find both relevant users as well as documents.

AU3 - They should be able to view all data held about them at any time, and delete it if they choose to.

AU4 - They should be able to build a network of friends which they can view.

AU5 - It should be easy for them to explore new and old content created by friended users.

Performance Requirements:

AP1 - Logging the user in should take a minimal amount of time (< 5 seconds)

AP2 - Adding and removing friends should be almost instantaneous (< 1 second)

AP3 - Browsing, viewing and sharing should be quick and responsive (update content < 5 seconds)

B - Data analysis and retrieval

Functional Requirements:

- BF1* - New documents must be parsed into the system and added to the server-side file structure.
- BF2* - The System must be capable of splitting the document into a list of all of its terms.
- BF3* - It must be able to “stem” the words to make them shorter as well as help to categorize terms.
- BF4* - The TFIDF of each term must be calculated and added to the vector space model.
- BF5* - The vector space model for all documents must be storable as a sparse matrix within the systems database.
- BF6* - The weights from each document must be retrievable from the database to reform the documents vector object.
- BF7* - The vectors of two objects in the system must be comparable using cosine similarity to compute the dimensional proximity.
- BF8* - Queries and profiles must also utilize the vector space model to their own vectors.
- BF9* - There must be a way for the system to exclude common terms which have no importance in the VSM.
- BF10* - Documents and Profiles must be removable from the system without having a large effect on the rest of the data except for recalculating term weightings.

User Requirements:

- BU1* - Terms that haven't been used for a period of time by the user must become less weighted over time to reflect the users changing interests, so that suggestions are timely relevant.
- BU2* - Vectors of high similarity must be selected by the system and viewable by the user.

C - Mobile Application

Functional Requirements:

- CF1* - Must be able to send and receive data to the remote server application over WAN and LAN using TCP protocol.
- CF2* - Must be able to invoke functions from the server to calculate VSM and store data.

- CF3* - Must be able to switch between user activities to access different views.
- CF4* - Must allow the user to logout of the app at any time.
- CF5* - Must feature scrollable text fields to show lists of items and large amounts of text.
- CF6* - Must feature clickable buttons that allow the user to complete a request.
- CF7* - Must have editable text fields allowing the user to input data.

User Requirements:

- CU1* - Must be able to validate user inputs, such as password and email fields.
- CU2* - Must allow the user to view Documents, Users and Personal Details.
- CU3* - Must allow the user to select images from their gallery to be inserted into the system.
- CU4* - Must notify the user once a command has been completed (such as saving a document)
- CU5* - Must be intuitively designed, reducing visual clutter to aid simplicity.
- CU6* - Must feature a helpful guide to using the application.
- CU7* - Must accurately label all input/output fields for easy recognition.
- CU8* - A back button that will return the user to the previous view.
- CU9* - Feature a distinctive logo on both the action bar and the app launcher symbol.
- CU10* - The app must be aesthetically professional and pleasing to the user.

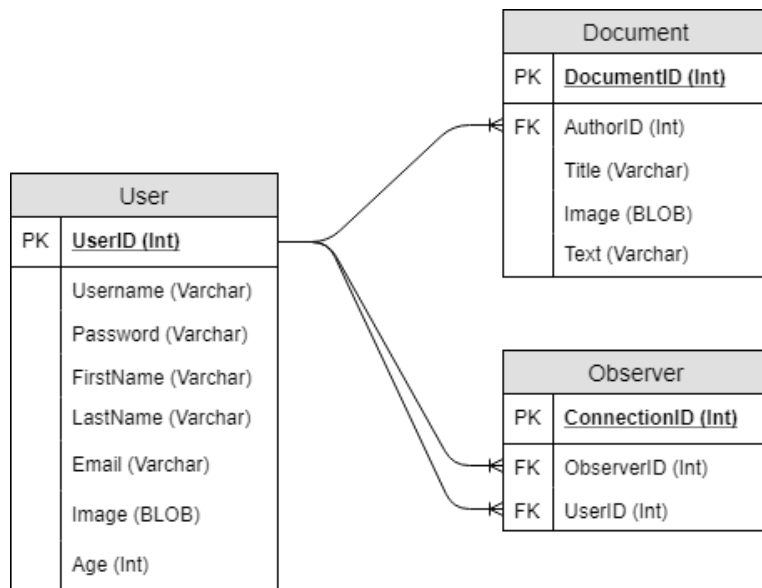
Performance Requirements:

- CP1* - Switching between views should be almost instantaneous (<1 second)
- CP2* - Notifications should be presented to the user as soon as a task has completed (<1 second)
- CP3* - Opening the application should take less than 10 seconds to load the home or login view.
- CP4* - The application should function consistently on all android devices about API build 15 (Ice cream sandwich)

Diagrams

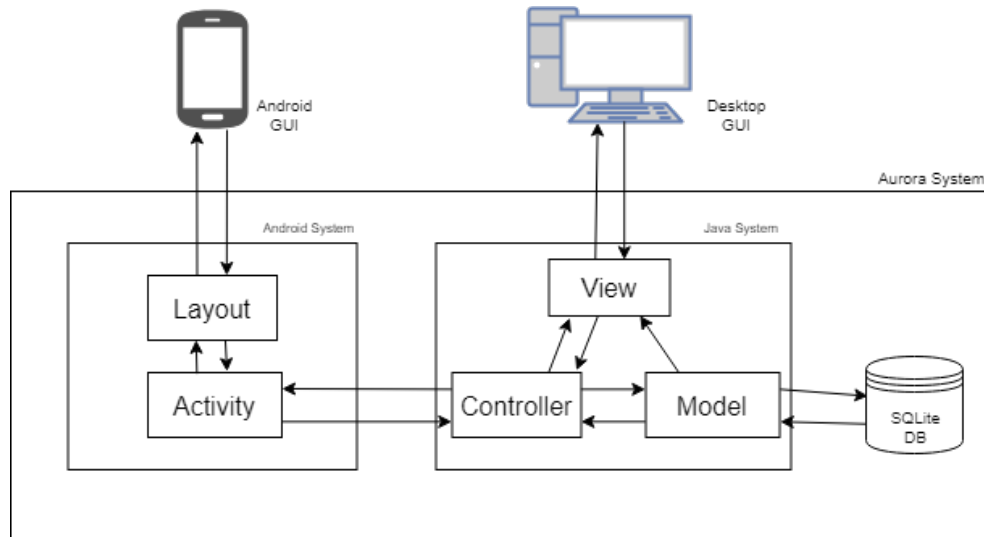
Entity Relationship Diagram

As a guideline for my system I will be using an ERD that represents the tables I will implement into my database, and how they will be related:



System Architecture Diagram

This diagram represents how I will be separating each part of my system, as well as how each of these parts will communicate with each other:



Project Lifecycle

I have divided my project into three phases to complete the project – I decided to do this so that I could prioritize the most important aspects first, so that the foundation of the systems architecture was always going to be implemented before any of the additional functionality which had lower dependencies. It also ensures that the product I produce at the end will have at least the minimum requirements I have detailed in my planning.

The three phases are the social platform which will cover all requirements in section A of my planning. This will allow for users to login and utilize the application to share and view content with other users. It will be the base of my project and should be able to layout the core functionality of the system. It will not have any dependencies; however, it will not have any visual model so I may design a simple desktop GUI that will accompany the application to test and demonstrate system behaviour.

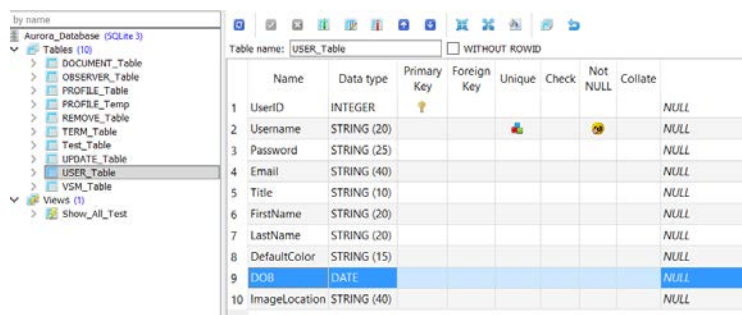
The next phase will be the Information retrieval and analysis which will implement the ability to process the document text into usable information that can be used to create complex queries and helpful suggestions for the user – this will rely heavily on the first phase being implemented as it will need to allow documents to be added and removed from the vector space model.

The last phase of my project will be the mobile application – this will be a front-end interface for users to interact with the implemented system. A large amount of the functionality will already be included in the main application so this should require the least number of components, however it will have the largest amount of dependencies and relies on having a stable connection with the rest of the system which may be hard to create.

Implementation

Phase 1 – Social Platform

I decided to first create my database and fill it with the tables that that I specified in my ERD diagram. I used the software SQLite along with the SQLite studio editor as they are open source and can be connected to java easily. The editor also allowed me to implement the tables visually without too much use of SQL at this point. Meaning I could quickly make the tables and make changes to it if I needed to, in my ERD I initially planned to include the users 'Age' in USER_Table however I changed this to storing the date of birth (DOB) instead as a Date field. This is because the age of the user will change each year and it will be easier to automate if date is stored.



	Name	Data type	Primary Key	Foreign Key	Unique	Check	Not NULL	Collate
1	UserID	INTEGER	Primary Key					NULL
2	Username	STRING (20)						NULL
3	Password	STRING (25)						NULL
4	Email	STRING (40)						NULL
5	Title	STRING (10)						NULL
6	FirstName	STRING (20)						NULL
7	LastName	STRING (20)						NULL
8	DefaultColor	STRING (15)						NULL
9	DOB	DATE						NULL
10	ImageLocation	STRING (40)						NULL

I then created a new java project using eclipse, it is the IDE I am most familiar with. I then created a new class called 'DBConnection' that will act as a bridge between my java application and the database. I wrote the following code which establishes a new connection every time the class is instantiate and will allow me to make queries and updates to the tables of my database at any time.

```
public class DBConnection {
    Connection conn = null;
    public static Connection dbConnector() {
        try {
            Class.forName("org.sqlite.JDBC");
            Connection conn = DriverManager.getConnection("jdbc:sqlite:resources\\Aurora_Database.sqlite");
            System.out.println("Connected to Aurora.. ");
            return conn;
        } catch (Exception e) {
            System.out.println(e);
            return null;
        }
    }
}
```

I created both a Document and Profile class, they will be created each time a USER row or DOCUMENT row is retrieved from the database to create a new object to represent the data. They also contain the corresponding getter and setter methods to retrieve the private class variables. The constructor for the document class. The constructor for both will take in the DocumentID or UserID as parameters so the object is easily distinguished; or the Text data from the document or the users username if the object has not been inserted into the database yet and assigned a unique identifier.

```

public class Profile implements Serializable {

    private static final long serialVersionUID = 9190933018758406998L;

    private int userID;
    private String username;
    private String password = "";
    private String firstName = "";
    private String lastName = "";
    private String defaultColor = "";
    private String title = "";
    private String email = "";
    private byte[] profileImage;
    private boolean isSelected = false;
    private int age;

    public class Document implements Serializable, Comparable<Document> {

        private static final long serialVersionUID = -7294652124482841060L;

        private int documentID;
        private String documentText;
        private String textLocation;
        private String imageLocation;
        private byte[] documentImage;
        private Profile author = new Profile();
        private int totalWords;
        private double similarity;
        private String lastUpdated;
        private String title = "Untitled";
    }
}

```

I then needed a couple of methods for which to easily query and update the database when necessary since I will be making a lot of SQL statements. These short methods will execute some input SQL and return a result set if applicable. It also throws an SQLException if the input string is invalid.

```

public ResultSet queryDatabase(String sql) {
    ResultSet rs = null;

    try {
        rs = stmt.executeQuery(sql);
    } catch (SQLException e) {
        e.printStackTrace();
    }

    return rs;
}

public boolean updateDatabase(String sql) {
    try {
        stmt.executeUpdate(sql);
        return true;
    } catch (SQLException e) {
        e.printStackTrace();
    }

    return false;
}

```

I next implemented a method for adding users to the database, to do this I created a new class that will act as a controller between the database and the java application. To send SQL statements an instance of the database connection must be either created or passed to the controller, from there a Statement object can be used to input SQL and receive a ResultSet containing data from the rows and columns produced from the SQL. To add a new user a new row must be inserted into the USER_Table containing all the data affiliate with each profile. I did the same thing for adding documents.

```

String sql = "INSERT INTO USER_Table (Username, Password, DOB, FirstName, LastName, DefaultColor, Title, Email, ImageLocation)"
    + " VALUES ('" + user.getUsername() + "', '" + user.getPassword() + "', "
    + user.getAge() + ", '" + user.getFirstName() + "', '" + user.getLastName()
    + "', '" + user.getDefaultColor() + "', '" + user.getTitle() + "', '" + user.getEmail() + "', '" + location + "')";

profileDB.updateDatabase(sql);

```

A big change I soon needed to make was I planned to include the images and text documents in the database since it supports string and BLOB datatypes. However, I realized that it would be impractical storing very long strings of text and large images, so instead I modified the database to include a reference to the file location of the corresponding files. That way they can still be accessed without loading them directly from the database. To implement this, I first needed to create a file

structure for the first time the program is run. If the directory does not exist, it will create it so that files can be put in the correct location.

```
private void formatFolders(Profile user) {
    String s = File.separator;
    ArrayList<String> dirs = new ArrayList<String>();
    String profileURL = System.getenv("APPDATA") + s + "Aurora" + s + "profile_data" + s + user.getUsername();
    dirs.add(profileURL);
    dirs.add(profileURL + s + "profile");
    profileURL = profileURL + s + "docs";
    dirs.add(profileURL);
    dirs.add(profileURL + s + "image");
    dirs.add(profileURL + s + "text");
    for(String dir : dirs) {
        createDir(dir);
    }
}
```

Next, I added an additional method to the database controller that takes the image file and stores it within the users unique file location (based on their username) so that it can be retrieved later. I decided to place all created files inside of APP_DATA since it doesn't require admin permissions to write and read from. The file location can be made using the function System.getenv(). This makes it easy for the program to be installed onto other machines.

```
public void setProfileImage(Profile user) {
    String s = File.separator;
    String location = s + theSession.getUser().getUsername() + s + "profile" + s + "icon.png";
    String url = System.getenv("APPDATA") + s + "Aurora" + s + "profile_data" + location;
    Image image = byteArrayToImage(user.getImage());
    try {
        ImageIO.write((RenderedImage)image, "png", new File(url));
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

I then created remove and update functions for both documents and users so that the data can be modified after it has been added to the database, for updating the SQL 'UPDATE' command can be used followed by which fields need to be changed and on what rows the changes should occur, similarly the DELETE command will remove the whole row from the table.

```
public void editString(String tbl, String idCol, String valCol, String val, int id) {
    sql= "UPDATE " + tbl + "_table SET " + valCol + "=" + val + " WHERE " + idCol + "=" + id;
    updateDatabase(sql);
}
```

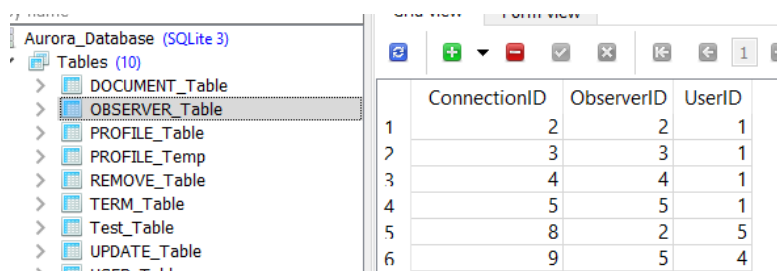
To allow the user to get all their documents I create a method that will search through the DOCUMENT_Table and select all documents which AuthorID is equal to the requested user. I did this by creating a new SELECT SQL statement. It returns a ResultSet containing each row of the query

results. Using a while loop I can extract the data from each line and add it to a new ArrayList of documents. The list can then be returned to the user.

```
public ArrayList<Document> getUserDocuments(Profile u) {
    sql = "SELECT * FROM DOCUMENT_Table WHERE AuthorID=" + u.getUserID() + " ORDER BY DocumentID";
    ResultSet rs = queryDatabase(sql);
    ArrayList<Document> documentList = null;
    documentList = createDocuments(rs);

    return documentList;
}
```

A big part of the social platform is allowing users to link their profiles together. To do this I created an "OBSERVER_Table" this is because when a user links their profiles to another they begin to 'Observe' the other user. The link is effectively a relationship between two UserID's one being the observer and the user being observed. A new link can be added by inserting the new connection to the table. And likewise, the connection between users can be broken by removing the row.



ConnectionID	ObserverID	UserID
1	2	1
2	3	1
3	4	1
4	5	1
5	8	5
6	9	4

To get a list of all observers or people observing a user, two new methods can be implemented. They will select all users affiliated with the requested UserID and return a list of profiles, similarly to requesting all the user documents. The list can then be returned to the user to view; this is done as both a list of observers and an observing list as shown below.

```
public ArrayList<Profile> getObserving(Profile u){
    sql = "SELECT * FROM OBSERVER_Table WHERE ObserverID=" + u.getUserID();
    ResultSet rs = queryDatabase(sql);
    ArrayList<Integer> userID = new ArrayList<Integer>();
    ArrayList<Profile> observers = new ArrayList<Profile>();
    Profile user;
    try {
        while(rs.next()) {
            userID.add(rs.getInt(3));
        }
        for(int id : userID) {
            user = getUser(id);
            observers.add(user);
        }
        return observers;
    } catch (SQLException e) {
        e.printStackTrace();
    }
    return null;
}
```

Additionally, to allow for simple searching I implemented a method that generates a new SQL statement that will use the 'LIKE' term to compare inputted strings to each user and return all usernames that contain the input string plus any characters after it. This will return a list of related users.

```
public ArrayList<Profile> getUserResults(String query) {  
    String sql = "SELECT * FROM USER_Table WHERE Username LIKE '" + query + "%'";  
    ResultSet rs = profileDB.queryDatabase(sql);  
    return profileDB.createProfiles(rs);  
}
```

Now that the core functionality is implemented I decided to test the system by adding a few users to the database and creating links between them. I could then successfully output documents, linked users and search results. For the next part I decided to build a simple desktop GUI that would make it easier to debug the system once I begin building the VSM.

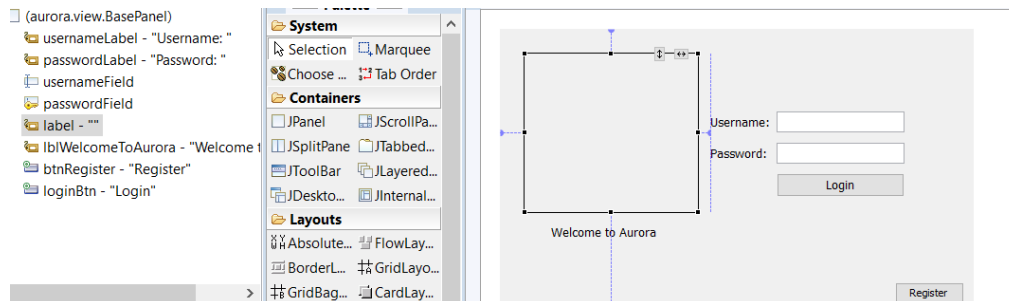
To create the GUI, I used swing since I am most familiar with the package, it is also compatible with the java window builder. The first task is to create a new JFrame which will contain the applications views which are represented as panels. To do this I created a new class that extends JFrame called BaseFrame which creates a new window that is 500x300, and will add a new login panel to the frame. From a launcher the BaseFrame can then be created to start a new window.

```
public class BaseFrame extends JFrame {  
    private LoginPanel loginPanel;  
    public BaseFrame(LoginController loginController) {  
        loginPanel = new LoginPanel(loginController);  
        setupFrame();  
    }  
    private void setupFrame() {  
        this.setContentPane(loginPanel);  
        this.setTitle("Aurora v1.0.0");  
        this.setSize(500, 300);  
        this.setVisible(true);  
    }  
}
```

To keep my project organized I used the MVC pattern, I started this by creating a new model view and controller class for the login screen. The model is linked to the database controller and will be used to link between the view controller and the database controller. It will also contain any other methods that are needed to process a user's request.

I decided to build the view using the java window builder, it is a helpful tool that allows you to create quick interfaces. The login page features a login button followed by two text fields where a user can input both their username and password. Additionally, a register button is added when I have implemented the registration form. To validate the login a new method is created that selects the

first row (since there should be at most one match) of a query for all users with a username and password that matches the parameters supplied. If one exists they can continue to login, if not then an output will simply tell them that either the username or password is invalid.



Once a user has successfully logged in they must be able to access their unique profile data. To ensure that this a new session must be started. I created a new session class that will be instantiated, it contains the user object as well as the connection that was made with the database; this is so the same connection does not have to be made every time the user changes view.

```
public class Session {
    private Profile theUser;
    private Connection theConnection;
    private SocketController clientSocket;

    public Session(Profile user, Connection conn) {
        this.theUser = user;
        this.theConnection = conn;
    }

    public Session login(Profile p) {
        profileDB.addUserLinks(p);
        theSession = new Session(p, conn);
        return theSession;
    }
}
```

To update the BaseFrame's current panel I implemented a method for each of the possible panels the user can go to from the login panel. I started with the home panel – it first creates a new home controller that will itself create a new HomePanel. Then it is set as the contentPane. The old contentPane (in this case the login panel) can then be added to the home controller as the previous view. This makes it easy in the future for the user to go back.

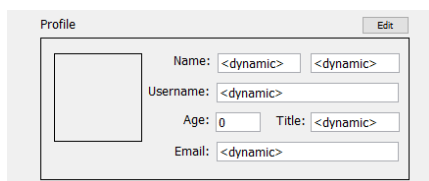
```
public void login() {
    HomeController hc = new HomeController(theSession);
    hc.setFrame(theFrame);
    hc.setPreviousPanel((BasePanel) theFrame.getContentPane());
    HomePanel hp = new HomePanel(hc);
    theFrame.setContentPane(hp);
    theFrame.pack();
}
```

Since a lot of the views will contain similar elements and to maintain a consistent look I created a new class called MainPanel that extends JPanel. It contains all the buttons and textfields for the actionbar at the top of the screen. It also has basic buttons such as the back button and logout button. I arrange the class of each of my panels to follow a similar structure. The constructor will do three things. It will initialize all the variables associate with an element on the panel, it will create

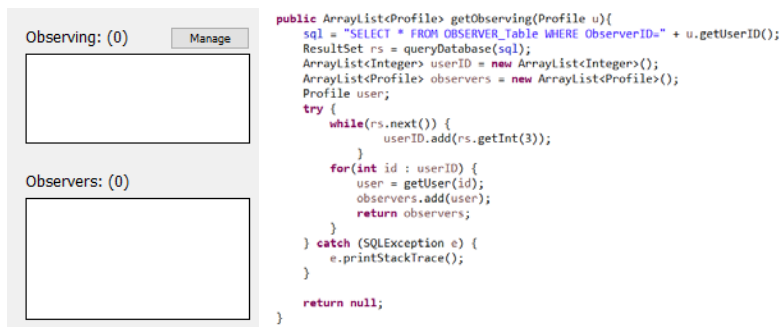
listeners for the elements that are interactable and it will input the data that the panel needs to obtain from the controller.

```
public void setupData() {
    Profile user = theController.getUser();
    sessionLabel.setText("Logged in as: " + user.getUsername());
}
```

The HomePanel contains a lot of elements. At the top of the panel the user details are displayed next to their profile picture. The user details are obtained by extracting the specific user from the database and getting each of the Profile objects fields. Additionally, the object is obtained by reading in the image from the specified image location. It is then set as the icon image of a JLabel. In the top corner of the details is an edit button this will send users to a separate panel which users can change the data stored about them.



Further down the page is a section for observers and observing, they contain all the profiles associated with this user. As this is a list of unknown length I created a JList object that is wrapped by a scrollpane, this ensures that if the length of the list is larger than the contain it will allow the user to scroll through hidden list items. The list is obtained by querying the database for all connections associated with that user. The JList requires an array as input, therefore I convert the ArrayList accordingly to produce an array of strings containing the username of each profile.



The list is focusable by default but has no value associated with it except for the username and the listIndex. Therefore, a separate list of users must be stored in the controller that will be used to obtain all the user details when they are selected – by cross referencing the list index to the array index. From there a button below will allow the user to interact with those profiles. For observers, they can press a view button to go to their specific profile. Users in the observing list can also be viewed as well as managed by pressing the “Manage” button which will open the manage panel.

```

public void viewObserver(int index) {
    if(index >= 0) {
        Profile profile = theModel.getObservers()[index];
        theModel.addUserLinks(profile);
        ProfileController pc = new ProfileController(theSession, profile);
        pc.setPreviousPanel((BasePanel) theFrame.getContentPane());
        pc.setFrame(theFrame);
        ProfilePanel pp = new ProfilePanel(pc);
        theFrame.setContentPane(pp);
        theFrame.pack();
    }
}

public void manageObserving() {
    ManageController mc = new ManageController(theSession);
    mc.setPreviousPanel((BasePanel) theFrame.getContentPane());
    mc.setFrame(theFrame);
    ManagePanel mp = new ManagePanel(mc);
    theFrame.setContentPane(mp);
    theFrame.pack();
}
}

```

The last section of the home page is the document list. It functions similar to the user lists however will display all documents that the user has made. Buttons below will allow the user to view, remove or add a new document. When a new document is added it is inserted into the database and appears in the list as "New post" when viewed it will show a blank document the user can edit.

Documents: (0)

View
Add
Remove

```

public void addUserDocument() {
    SimpleDateFormat formatter = new SimpleDateFormat("dd/MM/yyyy");
    Date date = new Date();
    Document doc = new Document("");
    doc.setTitle("New Post");
    doc.setAuthor(theSession.getUser());
    doc.setImageLocation("\\default\\docs\\image\\1.png");
    doc.setTextLocation("\\default\\docs\\text\\1.txt");
    doc.setLastUpdated(formatter.format(date));
    profileDB.addNewDocument(doc);
    theSession.getUser().addDocument(doc);
}

```

When viewing a user, a new ProfilePanel will be created and added to the frame. The panel will contain the details of the viewed profile and allow them to view the users and documents they are associated with. I came across a problem when trying to build this as when a user is created, the list of connected users is also tied to the profile, these profiles are also created and if they have connected users then it creates a recursive loop where it is continuously trying to obtain connected users. To solve the issue, I stopped the connected users from being retrieved when first creating the profile, until the specific user is being viewed. Although the profile panel is similar to the home panel, it does not contain any edit, add, remove buttons as viewed user details should not be changeable by another user.

Aurora Logged in as:

Profile Remove

Name: <dynamic> <dynamic>

Username: <dynamic>

Age: 0 Title: <dynamic>

Email: <dynamic>

Documents: (0) Observing: (0)

View

Back Log Out

```

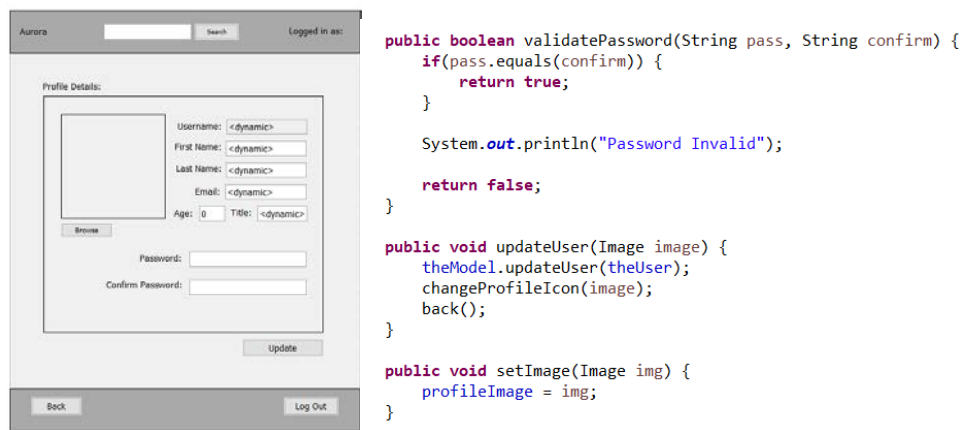
private void updateDetails() {
    Profile user = theController.getViewedUser();
    fldUsername.setText(user.getUsername());
    fldFirstName.setText(user.getFirstName());
    fldLastName.setText(user.getLastName());
    fldTitle.setText(user.getTitle());
    fldEmail.setText(user.getEmail());
    fldAge.setText(Integer.toString(user.getAge()));
}

```

The ManagePanel will display all the profiles that the active user is observing. This functions as a list like before allowing them to view each profile, however also includes a delete button. When clicked it will remove the user connection from the database and update the list to no longer contain the user, they will now no longer receive posts made by them.



The DetailPanel is accessed when the user presses the edit button from the home page. It looks similar to the home page containing all the user's details, however when they press the edit button at the bottom of the page each line of text becomes editable. The user can change the text (except for the username as this is unique to the account) to a different value.



The users profile picture can also be set by clicking on a button labelled "Browse". This creates a new window by initializing a new JFileChooser which will display the user's file explorer allowing them to select an image and return to the main frame. The new image will then replace the old one. When the user presses the Save button which becomes visible when the edit button is selected, it will create a new profile object and sent to the database controller. The controller will create a new update SQL statement which will modify the row within the USER_Table associated with the active profile. The panel will then change back to the home panel to confirm that the changes have been made.

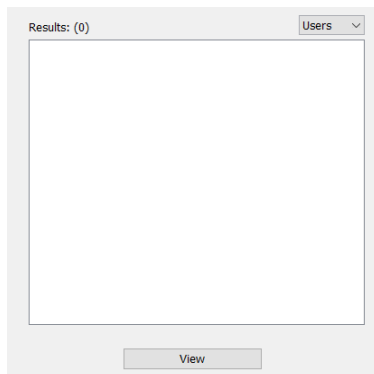
```

public void browse() {
    JFileChooser fileChooser = new JFileChooser();
    fileChooser.setCurrentDirectory(new File(System.getProperty("user.home")));
    FileNameExtensionFilter filter = new FileNameExtensionFilter("*.IMAGE", "jpg", "png", "gif");
    fileChooser.setFileFilter(filter);
    int result = fileChooser.showSaveDialog(null);
    if(result == JFileChooser.APPROVE_OPTION) {
        File selectedFile = fileChooser.getSelectedFile();
        try {
            profileImage = ImageIO.read(selectedFile);
            profileIcon.setIcon(new ImageIcon(profileImage.getScaledInstance(150, 150, Image.SCALE_DEFAULT)));
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

Within the actionBar of each panel is a textfield which users can use to search for users. To do this they can input a string and press the Search button which will send them to the SearchPanel. This will contain another JList with all the query results from the SQL Like statement that I constructed before, similar to the home panel they are able to select the user and then view their profile. In the

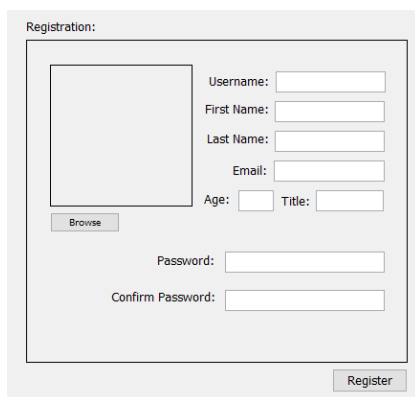
future I will be adding the functionality to search for documents as well, however this will require the VSM to be implemented.



```
public void search(String query) {
    SearchController sc = new SearchController(theSession, query);
    sc.setFrame(theFrame);
    sc.setPreviousPanel((BasePanel) theFrame.getContentPane());
    SearchPanel sp = new SearchPanel(sc);
    theFrame.setContentPane(sp);
    theFrame.pack();
}

public String[] getResultsUsername() {
    userList = theModel.getUserResults(theQuery);
    resultNumber = userList.size();
    String[] strArr = new String[resultNumber];
    for(int i = 0; i < resultNumber; i++) {
        strArr[i] = userList.get(i).getUsername();
    }
    return strArr;
}
```

Lastly, the RegisterPanel will allow for new users to sign up. It looks very similar to the DetailPanel, however each field will be blank as there is no previous user data. Once they have filled out each field, they can press the register button. This will automatically insert a new user into the database by creating a new SQL statement within the database controller and then send the user to the HomePanel with the created profile as the active user.



```
public void Register() {
    Profile registeredUser = theModel.registerUser(newUser);

    if(profileImage != null) {
        addProfileIcon();
    }

    theSession = theModel.createSession(registeredUser);
    HomeController hc = new HomeController(theSession);
    hc.setFrame(theFrame);
    HomePanel hp = new HomePanel(hc);
    theFrame.setContentPane(hp);
    theFrame.pack();
}
```

Phase 2 - Data Analysis and Retrieval

At this point I have created an application which acts as a simple social platform. I can now focus on implementing a vector space model and recommender system. The main portions of this phase will be finding a way to extract quantitative values from the systems document objects, allow users to search for documents using abstract terms from text and finally attach vector weightings to profiles so that appropriate documents will be suggested to the user.

To complete the first step, I had to create a way to parse the text from the stored file into a new document object. I created a new class that reads each line in and adds it to a string. This can then be set as the documents text.

```
public String read(String dir) throws Exception {  
    FileReader file = new FileReader(dir);  
    BufferedReader reader = new BufferedReader(file);  
  
    String text = "";  
  
    String line = reader.readLine();  
  
    while (line != null){  
        text += line;  
        line = reader.readLine();  
    }  
  
    return text;  
}
```

Because each term of the document must be analysed separately I next needed to find a way to split the text into individual words. To do this I created a new Word class that will store the string of each word. I then created a class called "Splitter" that will split the text where it encounters a space or a hyphen in the inputted string. The strings are then added to individual word objects and stored in a word ArrayList to be worked on further.

```
public ArrayList<Word> splitText(String s){  
    ArrayList<Word> wordList = new ArrayList<Word>();  
    String[] wordArr = s.replaceAll("[^a-zA-Z -]", "").toLowerCase().split("\\s+");  
    for(String word : wordArr){  
        Word w = new Word(word);  
        wordList.add(w);  
    }  
  
    return wordList;  
}
```

The next important step in the process is to shorten the words, and concatenate them into term roots; for example words like 'computer', 'computing' and 'computation' will be all changed to

‘comput’ this serves the purpose of reducing the amount of terms in the vector space model and also makes the documents more comparable, regardless if a document contains ‘computer’ or ‘computing’. The method I will be using to do this is called ‘porter stemming’ and I will be including my own implementation of this algorithm.

```
private String porter2(String s){
    String string = s;
    check = false;
    String[] changes = {"ational", "ate", "tional", "tion", "enci", "enc", "anci", "ance",
        "izer", "ize", "abli", "able", "alli", "al", "entli", "ent", "eli", "e", "ousli",
        "ous", "ization", "ize", "ation", "ate", "otor", "ote", "alism", "al", "iveness", "ive",
        "fulness", "ful", "ousness", "ous", "aliti", "al", "iviti", "ive", "biliti", "ble"};

    if(measure > 0){
        for(int i = 0; i < changes.length; i+=2){
            replacer = createReplacer(string, changes[i], changes[i+1]);
            string = replace(replacer, 8 + i);
        }
    }

    return string;
}
```

Porter stemming is broken into a series of steps that must be completed for each word. Natural language is very complicated and has a large number of rules and exceptions. I created a new class called WordStemmer to handle this process. A lot of the stemming relies on suffix replacements. In the first step any word ending in “sses” is replaced with “ss” for example; therefore, I created a way to automate this.

```
private String replace(SuffixReplacer sr, int i){
    if(!check){
        String string = sr.getReplaced();
        check = sr.Success();
        success[i] = check;
        return string;
    }
    return sr.getOriginal();
}

private SuffixReplacer createReplacer(String string, String suffix, String replacement){
    return new SuffixReplacer(string, suffix, replacement);
}
```

Once the words have been reduced to stems, the term frequency can be calculated. The term frequency is important as it is the first part of the TF-IDF calculation. The Term frequency is the amount of times the stemmed word appears in a specific document. I created a new class “FrequencyAssigner” it takes an ArrayList of words and creates a new ArrayList of TextWords,

TextWords are just an extension of Words that stores the word frequency additionally. Each word from the original list is compared to all the words in the new TextWord list. If there is a match then the frequency of the matching term is increased by 1 in the TextWord list and the original word is discarded. If there is no match then the old word is added to the TextWord list as a new TextWord object with a frequency of 1. Once all words from the original list have been iterated through, the new list will contain the frequency of each word with no repeating terms.

```
public ArrayList<TextWord> initiate(ArrayList<Word> w){
    int wordIndex;
    double scale;
    WordStemmer stemmer = new WordStemmer();
    ArrayList<TextWord> textWordList = new ArrayList<TextWord>();
    for(Word word : w){
        wordIndex = -1;
        word.setWord(stemmer.Stem(word.getWord()));
        for(TextWord textWord : textWordList){
            if(word.getWord().equals(textWord.getWord())){
                wordIndex = textWordList.indexOf(textWord);
            }
        }
        scale = 1;
        if(wordIndex != -1){
            textWordList.get(wordIndex).addTextFrequency(scale);
        } else {
            textWordList.add(new TextWord(word, scale));
        }
    }
    return textWordList;
}
```

Having the frequency done, the next part of the calculation is obtaining the document frequency, as this will regulate the weighting of terms that are not unique to the text. To do this the VSM_Table (explained further down) can be queried for all documents associated with a specific termID, the count() of these documents will reveal the document frequency for each term. The last part is to add 1 to the result, as the current document being added is not yet in the system; so it will needed to be included in the calculation.

```
protected int getDocFrequency(int termId) {
    String sql = "SELECT Count(*) FROM VSM_Table WHERE TermID=" + termId;
    ResultSet rs = queryDatabase(sql);
    int freq = 0;
    try {
        freq = rs.getInt(1);
    } catch (SQLException e) {
        e.printStackTrace();
    }
    return freq;
}
```

The last parts of the calculation are the total number of documents which can be obtained by simply finding the count of the entire DOCUMENT_Table – As well as the number of terms in the document being added, which is just the size of the TextWord list that was inputted. The total documents is important as it provides a scale for how common a word is, as the text database could be of any size.

Additionally, the amount of terms is used to normalize the text frequency – as for a similar reason the document could be of any size.

```
protected int getNumberOfDocs(){
    String sql = "SELECT COUNT(DocumentID) FROM DOCUMENT_Table";
    ResultSet rs = queryDatabase(sql);
    int total = 0;

    try {
        rs.next();
        total = rs.getInt(1);
        rs.close();
    } catch (SQLException e) {
        e.printStackTrace();
    }
    return total;
}

protected int getDocWordCount(int id) {
    String sql = "SELECT WordCount FROM DOCUMENT_Table WHERE DocumentID=" + id;
    ResultSet rs = queryDatabase(sql);
    int count = 0;
    try {
        while(rs.next()) {
            count = rs.getInt(1);
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }
    return count;
}
```

To finally put together the TFIDF score for each term, each of the above components come together in this equation:

$$\text{TFIDF} = (\text{termFrequency} / \text{totalTerms}) * \log(\text{totalDocuments} / \text{documentFrequency})$$

After some testing I decided this formula produced the most effective outcomes. I tested it by inputting a list of test documents containing very different content and output the words that had the highest TFIDF score for each document. I eventually produced precise results after some trial and error. Having calculated the TFIDF, the new document–term relationship can be added to the VSM_Table which holds the associated document and term, as well as the term frequency in that document and the TFIDF score.

```
public double calculateTFIDF(int termId, int documentId, double termFreq, int docChange) {

    int totalDocuments = getNumberOfDocs();
    double termFrequency = termFreq;
    int documentFrequency = getDocFrequency(termId) + docChange;
    int totalDocumentTerms = getDocWordCount(documentId);

    double TFIDF = (termFrequency / totalDocumentTerms) * (Math.Log(totalDocuments / (1 + documentFrequency)));

    if(TFIDF >= 0 && TFIDF <= 5 ) {
        TFIDF = BigDecimal.valueOf(TFIDF).setScale(4, RoundingMode.HALF_UP).doubleValue();
    } else {
        TFIDF = 0;
    }
    return TFIDF;
}
```

There is one thing else to consider when inserting a new document into the VSM; the document frequency for each inserted term in the associated documents will also increase by 1. Therefore, the TFIDF must be calculated again for every affected document term. To do this the list of documents

created when querying for the original documentFrequency can be used. Every row contained in the list must be updated within the VSM_table with a new TFIDF using the new frequency score.

```
private void addToTermAxis(String term, TextWord n, Document doc){
    int termId = getTermId(term);
    vsmInserts.add("(" + termId + ", " + doc.getID() + ", " + n.getTextFrequency() + ", " + calculateTFIDF(termId, doc.getID(), n.getTextFrequency()) + ")");
    ArrayList<Integer> documents = getRelatedDocs(termId);
    for(int docId : documents) {
        double TFIDF = calculateTFIDF(termId, docId, getTermFrequency(termId, docId), 1);
        vsmUpdates.add("(" + termId + ", " + docId + ", " + getTermFrequency(termId, docId) + ", " + TFIDF + ")");
    }
}
```

An issue I ran into was that whenever a new document is added into the table roughly 200 terms must be added/updated. This affects a large number of other rows meaning 2000+ SQL statements must be passed to the database. Each insert or update can take up to 40 milliseconds which can produce a 30 second delay in the system, despite everything else working efficiently.

```
private void updateDocVector(int docId, int termId, TextWord newTerm){
    double termFreq = newTerm.getTextFrequency();
    vsmInserts.add("(" + termId + ", " + docId + ", " + termFreq + ", " + calculateTFIDF(termId, docId, termFreq, 1) + ")");
}
```

My solution to this was to create a new table called the UPDATE_Table that is temporarily populated with all the rows being updated any changes that need to be made. I did this as SQLite allows for any number of rows to be inserted at once, however updates are limited to one at a time. With UPDATE_Table populated a single UPDATE statement can alter every row in the original VSM_Table which has an affiliated row in the UPDATE_Table. This brings down a 30 second action down to roughly 100 milliseconds which is a big improvement despite the added complexity.

```
public void batchVSMUpdate() {
    if(vsmUpdates.size() > 0) {
        String sql = "INSERT INTO UPDATE_Table VALUES ";
        for(int i = 0; i < vsmUpdates.size() - 1; i++) {
            sql = sql + vsmUpdates.get(i) + ", ";
        }
        sql = sql + vsmUpdates.get(vsmUpdates.size() - 1);
        updateDatabase(sql);

        sql = "UPDATE VSM_TABLE SET TFIDF = (SELECT TFIDF from UPDATE_Table WHERE TermID = VSM_TABLE.TermID AND DocumentID = VSM_TABLE.DocumentID)";
        updateDatabase(sql);
        sql = "DELETE FROM UPDATE_Table";
        updateDatabase(sql);
        vsmUpdates = new ArrayList<String>();
    }
}
```

The opposite case is where a document is removed from the VSM. It will be very similar to adding, however there are some extra checks that need to be made. Like before, the documents text will again be split into frequency assigned TextWords. Firstly, every row in the VSM_Table that is associated with the document being removed is deleted. Then the document frequency is obtained

from the database - If it is equal to 0; that means the term is no longer in the VSM and can therefore be deleted from the TERM_Table to save space.

```
public void remove(Document doc){
    ArrayList<Term> oldTerms = getDocTerms(doc.getID());
    String sql = "DELETE FROM VSM_Table WHERE DocumentID=" + doc.getID();
    updateDatabase(sql);

    private void checkTerm(Term oldTerm){
        int docFreq = getDocFrequency(oldTerm.getId());
        if(docFreq <= 0) {
            termDeletes.add(String.valueOf(oldTerm.getId()));
        } else {
            updateTermAxis(oldTerm.getId());
        }
    }

    private void updateTermAxis(int termId){
        ArrayList<Integer> documents = getRelatedDocs(termId);
        for(int docId : documents) {
            double TFIDF = calculateTFIDF(termId, docId, getTermFrequency(termId, docId), 0);
            vsmUpdates.add("(" + termId + ", " + docId + ", " + getTermFrequency(termId, docId) + ", " + TFIDF + ")");
        }
    }
}
```

If it is not equal to 0; then every case a document is associated with that term, the TFIDF must be recalculated. Like before it will get a list of all the rows to update and then update them all at once. Since the removal of the document has already taken place, one does not need to be taken away from the frequency. The VSM will now no longer contain the document, or any terms in it that were unique.

```
public void batchTermDelete() {
    if(termDeletes.size() > 0) {
        String sql = "DELETE FROM TERM_Table WHERE TermID in (";
        for(int i = 0; i < termDeletes.size() - 1; i++) {
            sql = sql + termDeletes.get(i) + ", ";
        }
        sql = sql + termDeletes.get(termDeletes.size() - 1) + ")";
        updateDatabase(sql);
    }
    termDeletes = new ArrayList<String>();
}
```

My initial plan was to include a separate class that would handle the case in which a document is being updated. The idea was to remove words that were no longer present in the text and add ones that were. However, to do this it is necessary to find the symmetric difference between the two ArrayLists of old terms and new terms and then to distinguish between which ones are being added and removed. The process is complex and takes too much time to operate when working with more than 50 terms at a time. Therefore, I simplified the case of updating the document by simply making a new method that will sequentially remove the documents from the VSM (which contains the old terms) and then adds them all again (containing the new terms). It is a less elegant approach as a certain portion of the words will have been unchanged. However, It will still take less time as not as many checks need to occur.

```
public void updateDocument(Document doc) {
    remover.remove(doc);
    inserter.add(doc);
}
```

The VSM_Table now has a way for documents to be added, removed and updated, however there now needs to be a way to reconstruct the vector for each document whenever a comparison needs to occur. To do this I need to get a list of all the terms from the term table, check if the term is in the document from a list of document Terms and then get the TFIDF for that association if required,

otherwise it will be 0. A new double array will store the vector and can now be used for cosine similarity.

```
public double[] getDocumentVector(int docId) {
    ArrayList<Term> allTerms = getAllTerms();
    ArrayList<Term> docTerms = getDocTerms(docId);
    double[] vector = new double[allTerms.size()];
    Comparator<Term> c = new Comparator<Term>() {
        public int compare(Term a, Term b) {
            return a.getString().compareTo(b.getString());
        }
    };
    int index;
    for(int i = 0; i < allTerms.size(); i++) {
        index = Collections.binarySearch(docTerms, allTerms.get(i), c);
        if(index >= 0) {
            vector[i] = getTFIDF(docTerms.get(index).getId(), docId);
        } else {
            vector[i] = 0;
        }
    }
    return vector;
}
```

The cosine similarity will be calculated within a class called CosineSimilarity; it also requires a new class called Vector as the arrays must be handled as such. The Vector object will store the array as a variable. Cosine Similarity requires the normal of two vectors as well as the dot product. The normal is calculated by adding together each term to the power of 2, then square rooting the sum. The dot product can be calculated by multiplying together all term's TFIDF from each vector by the other term's TFIDF.

```
public class Vector {
    private double[] weights;

    public Vector(){
    }

    public Vector(double[] d){
        weights = d;
    }

    public double norm(){
        double norm = 0;
        for(double TFIDF: weights){
            if(TFIDF > 0){
                norm = norm + Math.pow(TFIDF, 2);
            }
        }
        return Math.sqrt(norm);
    }

    public double dotProduct(Vector v2){
        double product = 0;
        for(int i = 0; i < weights.length; i++){
            if((weights[i] > 0) && (v2.getWeights()[i] > 0)){
                product += (weights[i] * v2.getWeights()[i]);
            }
        }
        return product;
    }
}
```

The CosineSimilarity class will take the two documents, get their vsm arrays, pass them into new vector objects, get their normal and dot product and then finally calculate the Cosine Similarity. This is done by dividing the dot product by the two normals multiplied together. This will return a double

between 0 and 1 and reflects how similar the documents are in terms and the frequency of those terms. The class can also handle other objects other than documents, as long as they can be formed as a vector of the VSM as explained below.

```
public double equate(Document d, Document d2){
    vector1 = createVector(d2.getVSM());
    vector2 = createVector(d.getVSM());

    dotProduct = vector1.dotProduct(vector2);
    vector1Norm = vector1.norm();
    vector2Norm = vector2.norm();

    similarity = dotProduct/(vector1Norm*vector2Norm);

    return similarity;
}
```

Documents can now be compared; however, the user doesn't want to make a new document every time they search for new content. Instead a string query must be converted into a vector using the VSM. To do this I made a class VSMQuery – it has a new method calculate that will take in a string convert it into an ArrayList of TextWord again and then begin to work out the TFIDF for each term.

```
public double[] calculate(String query){
    int index;
    ArrayList<Word> splitWords = splitter.splitText(query);
    ArrayList<TextWord> textWords = assigner.initiate(splitWords);
    int total = textWords.size();
    ArrayList<Term> allTerms = getAllTerms();

    Collections.sort(textWords, c);

    double[] weights = new double[allTerms.size()];

    for(int i = 0; i < allTerms.size(); i++) {
        index = Collections.binarySearch(textWords, allTerms.get(i).getString());
        if(index >= 0) {
            weights[i] = calculateQueryWeight(allTerms.get(i).getId(), textWords.get(index).getTextFrequency(), 1, total);
        } else {
            weights[i] = 0;
        }
    }

    return weights;
}
```

Each word in the query is compared to each term in the TERM_Table, if there is a match, then the document frequency will be the number of times that term appears in the VSM_Table (Not including the query instance). This can then be used to calculate the TFIDF and it is added to an array list of matching terms. If a query word does not match a term in the VSM then it is simply discarded as it will be of no use in finding similarity to other documents.

```
private double calculateQueryWeight(int termId, double termFreq, int docChange, int totalWords){
    int totalDocuments = getNumberOfDocs();
    double termFrequency = termFreq;
    int documentFrequency = getDocFrequency(termId) + docChange;
    int totalDocumentTerms = totalWords;
    double TFIDF = (termFrequency/totalDocumentTerms)*(Math.log(totalDocuments/(1 + documentFrequency)));
    if(TFIDF >= 0 && TFIDF <= 200 ) {
        TFIDF = BigDecimal.valueOf(TFIDF).setScale(4, RoundingMode.HALF_UP).doubleValue();
    } else {
        TFIDF = 0;
    }
    return TFIDF;
}
```

A new ArrayList is filled with terms present in the VSM, that can then be used to create a double array holding the queries vector. This is done by again checking each term in the TERM_Table for when the terms match it will add the TFIDF of the queries term to the vector array and otherwise it

will be set to 0. This will form a new vector that does not impact the system, however contains all the information necessary to compare it to other vectors.

```
public void setQuery(String query) {
    queryVector = new Vector(vsm.getQueryVector(query));
}

public double equate(Document d) {
    Vector v1 = new Vector(vsm.getDocumentVector(d));
    Vector v2 = queryVector;

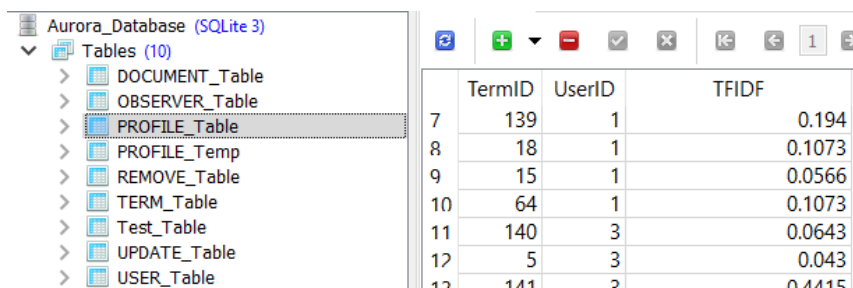
    dotProduct = v1.dotProduct(v2);

    similarity = dotProduct/(v1.norm()*v2.norm());

    return similarity;
}
```

The last part of phase two is to use the VSM to implement a recommender system that will suggest documents with high relevance to the user. To do this the profile will be treated as another vector in the system. It will have to be stored in a separate table however, as it will not influence the document VSM, however, it will need to be stored and modified when necessary. So, it cannot be disposable like the query vector

To do this I first created a PROFILE_Table, this will simply store the user, the term the user is associated with and the weighting of the word. (Very similar to the TFIDF in most cases but will be modified to reflect the user's current interests). The profile vector will be affected whenever the user views, adds, updates or removes a document.



	TermID	UserID	TFIDF
7	139	1	0.194
8	18	1	0.1073
9	15	1	0.0566
10	64	1	0.1073
11	140	3	0.0643
12	5	3	0.043
13	141	2	0.4415

Adding and viewing a document will be the same except adding a document will have a greater effect on the overall weighting, since it is more telling of a user's preferences. To do this it first gets the document being interacted with, splits it up into TextWord's just as the queries were. However,

a list of the profiles current associated terms is checked against each of the terms in the viewed/added document.

```
public void add(Document doc, Profile user) {
    ArrayList<Word> words = splitter.splitText(doc.getText());
    ArrayList<TextWord> textWords = assigner.initiate(words);
    ArrayList<Term> userTerms = getUserTerms(user.getUserID());

    updateProfile(doc, user, textWords, userTerms);

    batchProfileInsert();
    batchProfileUpdate();
}
```

If the term already exists, then the weighting is simply updated as the original TFIDF plus 10% of the terms TFIDF in this new document. This ensures that if the user views/adds similar content, the weighting for those terms will become stronger overtime. If there is no match to the word, then it is a new term for the profile to be matched with, and therefore it will be inserted into the PROFILE_Table with the documents TFIDF as the weighting.

```
public void updateSimilarWords(int docId, int userId, ArrayList<TextWord> words) {
    for(TextWord word : words) {
        int termId = getTermId(word.getWord());
        double tfidf = calculateTFIDF(termId, docId, word.getTextFrequency(), 0);
        userUpdates.add("(" + termId + ", " + userId + ", " + tfidf + ")");
    }
}
```

For the words that are already associated with the profile, before the document was added/viewed, if they are not present in the new ArrayList of terms, then they must become lighter in the profiles vector. This reflects that the user has lost interest in that term. To do this 0.03 is removed from the terms original weighting. If the new weighting is now less than 0 then, to save space, the term association within the PROFILE_Table is removed entirely. It will only be added again if the user views/adds a new document that contains that term.

```
public void updateOldTerms(int docId, int userId, ArrayList<Term> terms) {
    for(Term term : terms) {
        double tfidf = getUserTFIDF(term.getId(), userId) - 0.03;
        if(tfidf <= 0) {
            String sql = "DELETE FROM PROFILE_Table WHERE TermID=" + term.getId() + " AND UserID=" + userId;
            updateDatabase(sql);
        } else {
            userUpdates.add("(" + term.getId() + ", " + userId + ", " + tfidf + ")");
        }
    }
}
```

For the case that the user deletes one of their documents, it will attempt to reverse the effects of adding the document in the first place. As the weightings change overtime this may not remove it entirely, however it will impact the weighting of each present term in the deleted document. To do this all words in that document are compared to terms in the PROFILE_Table, if they exist then their

weights are reduced by the documents TFIDF score for that term. As before, if the terms weighting is then below 0 then the term will be deleted entirely from the PROFILE_Table.

I had a similar issue as adding documents as many rows will need to be updated at one time, so a similar batch update method is implemented to populate a new temporary table called PROFILE_Temp containing all the modified rows. It will then update the rows all at once.

```
public void batchProfileUpdate() {
    if(userUpdates.size() > 0) {
        String sql = "INSERT INTO PROFILE_Temp VALUES ";
        for(int i = 0; i < userUpdates.size() - 1; i++) {
            sql = sql + userUpdates.get(i) + ", ";
        }
        sql = sql + userUpdates.get(userUpdates.size() - 1);
        updateDatabase(sql);

        sql = "UPDATE PROFILE_Table SET TFIDF = (SELECT TFIDF from PROFILE_Temp WHERE TermID = PROFILE_Table";
        updateDatabase(sql);
        sql = "DELETE FROM PROFILE_Temp";
        updateDatabase(sql);
        userUpdates = new ArrayList<String>();
    }
}
```

For the user to now receive the suggestions, its vector is formed by first getting all terms that the profile is associated with from the PROFILE_Table, it then checks to see if that list contains each of the words in the TERM_Table. If it does then the weight will become the profile's term's weight otherwise it will be set to 0 and added to a new Boolean array. The array can then be used as a vector object as before to efficiently return a list of all the documents that have the highest cosine similarity to the profiles unique vector.

```
public double[] getProfileVector(int userId) {
    ArrayList<Term> allTerms = getAllTerms();
    ArrayList<Term> userTerms = getUserTerms(userId);
    double[] vector = new double[allTerms.size()];

    Collections.sort(userTerms, c);

    int index;
    for(int i = 0; i < allTerms.size(); i++) {
        index = Collections.binarySearch(userTerms, allTerms.get(i), c);
        if(index >= 0) {
            vector[i] = getUserTFIDF(userTerms.get(index).getId(), userId);
        } else {
            vector[i] = 0;
        }
    }
    return vector;
}

public double equateProfile(Document d) {
    Vector v1 = new Vector(vsm.getDocumentVector(d));
    Vector v2 = userVector;

    dotProduct = v1.dotProduct(v2);

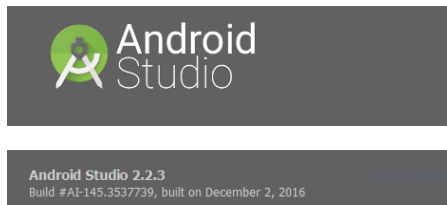
    similarity = dotProduct/(v1.norm()*v2.norm());

    return similarity;
}

public void setProfile(Profile u) {
    userVector = new Vector(vsm.getProfileVector(u));
}
```

Phase 3 – Mobile Application

The last part of my project is to create an android mobile application that can be used to communicate with the system remotely. I will be using the Android SDK along with android studio to develop this part of the project as it will be the most practical and recommended software to build a mobile application.



Before I could properly begin building the app, I had to create a connection to the java system. My first idea was to use the standard remote method invocation (RMI) package with java, however the android sdk does not support RMI by default. I could use an external package such as 'LipeRMI' however I was uncomfortable with using it, and it wouldn't provide all the functionality I would need for 2-way communication.

As a result, I decided to go with a socket approach instead, they would allow me to send data to the output stream and receive data from the input stream of both parts of the system. I first created a class in the java program called AuroraServer. This class will have two methods Run and Shutdown. The Run method will create a new ServerSocket on a determined port, then it will start a while loop that will effectively listen for any new connection requests from a client socket whilst the server is active.

```
public void run() {
    isRunning = true;
    Socket socket = null;
    System.out.println("Aurora: Starting..");
    try {
        serverSocket = new ServerSocket(port);
    } catch (IOException e) {
        e.printStackTrace();
        return;
    }
    VectorSpaceModel vsm = new VectorSpaceModel(DBConnection.dbConnector());
    System.out.println("Aurora: Listening..");

    while (isRunning) {
        try {
            socket = serverSocket.accept();
            new SocketController(socket, vsm).start();
        } catch (IOException e) {
            if(!e.getClass().equals(SocketException.class)) {
                e.printStackTrace();
            }
        }
    }
    return;
}
```

When a new connection has been made an instance of SocketController is created and passed to the clients socket as well as the VSM as this will be shared between all clients. The SocketController will be explained further down. The ServerSocket will again begin to listen out for more clients trying to connect to the systems. The Shutdown method will simply set the Boolean variable isActive to false and close the server socket, at this point no more connections can be made and all client sockets will also be closed.

```
public void shutdown() {
    System.out.println("Aurora: Shutting down..");
    isRunning = false;
    try {
        serverSocket.close();
        System.out.println("Aurora: Shutdown complete.");
    } catch (IOException e) {
        e.printStackTrace();
        System.out.println("Aurora: Shutdown failed.");
    }
}
```

On the android side, I created a similar class called AuroraClient; it also has a Run and Shutdown method. The Run method will create a new Socket object, entering in the IP of the ServerSocket (which will be whichever machine the server is run from) as well as the server sockets port – the port will direct where the TCP request must be directed to once it reaches the appropriate machine. The port for the server and client must be identical to create a connection.

```
public class AuroraClient extends Thread {

    private final String ip = "192.168.0.11";
    private final int port = 10777;
    private boolean isConnected = false;
    private Socket clientSocket = null;
```

```
public void run() {
    Socket socket = null;
    System.out.println("Client: Connecting..");

    try {
        clientSocket = new Socket(ip, port);
        System.out.println("Client: Connected.");
    } catch (IOException e) {
        e.printStackTrace();
        return;
    }

    isConnected = true;
    return;
}
```

The client will only have one connection to a server socket at once, therefore it is not necessary for the run method to listen to other sockets it sets the active Boolean to true and can now use the socket to communicate. It does however have a Shutdown method that will be set the active Boolean to false and will close the client socket. When this happens the server socket will throw an IOException and will signal that the client connection has been lost.

```
public void shutdown() {
    System.out.println("Client: Disconnecting..");
    isConnected = false;
    try {
        clientSocket.close();
        clientSocket = null;
        System.out.println("Client: Disconnected.");
    } catch (IOException e) {
        e.printStackTrace();
        System.out.println("Client: Disconnect Failed.");
    }
}
```

A connection can now be made. However, all the sockets will ever see is incoming and outgoing data in the form of primitives or Serialized objects. The way the system must operate is that the android application will invoke methods from the main application to add details to the database, recalculate the VSM, etc. This is very hard if it is only receiving arbitrary Strings and Integers, My solution to this was to build invoker classes.

The invokers are on the server side and work nearly identically to the panel controllers for the GUI, however they will accept Request and Response objects. These are new classes, they have a method identification number, and some data associated with it. The idea is to send some data to the server which will do something specific to it depending on which method the client wants to invoke. For an example the mobile application wants to get all the user details, so they will send the UserID wrapped in a request object with the method ID that will get the user from the database and return the object to the client wrapped in a response object.

```
public Response handleRequest(Request request) {
    int index = request.getMethod();
    Response response = new Response();

    switch(index) {
        case 0 : logout();
                response.setSwitch(true);
                break;
        case 1 : back();
                response.setSwitch(true);
                break;
        case 2 : String query = (String)request.getObject();
                search(query);
                response.setSwitch(true);
                break;
    }

    return response;
}
```

The SocketController's job is to handle the clients socket, when it receives a request object it will send it on to the active invoker, once the invoker has handled the clients request, it will send a response object back to the controller which sends it back to the client. The SocketController also holds the Session object associated with that client as well as the servers VSM, meaning the invoker can use both when necessary.

```
public void run() {
    remoteInvoker = new LoginInvoker(this);
    System.out.println("Client " + clientSocket.getInetAddress() + " connected.");

    while (isConnected) {
        try {
            objectOut = new ObjectOutputStream(clientSocket.getOutputStream());
            objectIn = new ObjectInputStream(clientSocket.getInputStream());
            try {
                request = (Request)objectIn.readObject();
                response = remoteInvoker.handleRequest(request);
            } catch (ClassNotFoundException e) {
                e.printStackTrace();
            }

            if ((request == null)) {
                isConnected = false;
            } else {
                objectOut.writeObject(response);
                objectOut.flush();
            }

        } catch (IOException e) {
            isConnected = false;
        }
    }
    System.out.println("Client " + clientSocket.getInetAddress() + " disconnected.");
    return;
}
```

I can now begin to build the android application. The process will be very similar to building the desktop GUI however with the added step of sending and receiving the request and response objects. It should be mentioned that the socket is handled on a separate thread to the main activity. This is so that whilst the socket is waiting for a response, it doesn't freeze the program, this is done by creating a new AsyncTask, the task will take in a request and return with a response when it has finished.

```

protected class ServerRequest extends AsyncTask<Request, Void, Response> {

    Socket clientSocket = SocketHandler.getSocket();

    @Override
    protected Response doInBackground(Request... params) {
        ObjectOutputStream objectOut = null;
        ObjectInputStream objectIn = null;
        Request request = params[0];
        Response response = null;

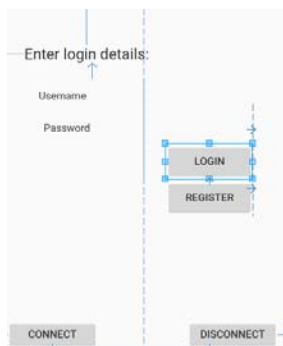
        try {
            objectOut = new ObjectOutputStream(clientSocket.getOutputStream());
            objectIn = new ObjectInputStream(clientSocket.getInputStream());
        } catch (IOException e) {
            e.printStackTrace();
        }

        try {
            objectOut.writeObject(request);
            response = (Response) objectIn.readObject();
        } catch (IOException e) {
            e.printStackTrace();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }

        return response;
    }
}

```

The Activities control the functionality of the app, they will be connected to a layout which is an .xml file which contains the elements and structure of each view – a visual editor can be used to create layouts. I first created the login activity and layout. This contains two EditText views which allow the user to input a string and store it within the element. It also has four buttons, for login and registration as well as connecting and disconnecting to the server socket.



The login button will check the contents of the EditText fields and send a server request to validate if the user login details are correct. If they are then a Boolean true will be sent back and the server-side invoker will switch to the home invoker. Meanwhile, the app will begin the home activity, this will change the view to feature the home layout. Alternatively, the register button will send the user to the register activity. Lastly, The connect and disconnect buttons are there to test and ensure a connection has been successful created or destroyed.

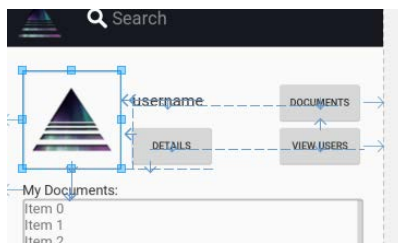

```

public boolean validateUser(String username, String password){
    ArrayList<String> login = new ArrayList<>();
    login.add(username);
    login.add(password);
    Request req = new Request(3, login);
    return serverRequest(req).getSwitch();
}

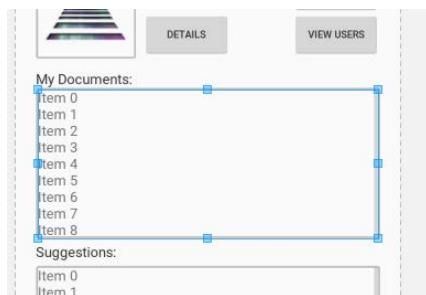
public void login(String username, String password){
    if(validateUser(username, password)){
        Request req = new Request(4);
        serverRequest(req);
        startActivity(new Intent(this, HomeActivity.class));
    } else {
        Toast.makeText(this, "Username or password incorrect", Toast.LENGTH_SHORT).show();
    }
}
}

```

The home panel contains the current users profile picture that is obtained by sending the profile object to the client. Since java and android have different Image data types it was hard to send/receive images. To deal with this I stored the image as a byte array and then converted the array into an image once it reaches its destination. It is also accompanied by some of the user's details.



To display the list of documents and/or profiles I create a new RecyclerView. What this does is creates a finite number of list views and recycles them whenever they go off the screen. This means that only the items actually being viewed will be rendered which improves the performance especially when dealing with images. Since my output from the server will be an ArrayList of profiles or documents I needed to convert them into a useable format by the RecyclerView.



To format custom data, a class that extends Adapter is required along with an item layout. The item layout will display data from each list item that is specified in the adapter. Since I need it to take either a list of documents or profiles I created different holder views which will handle the two objects in different ways.

```

public ItemAdapter(Context context, ArrayList<Profile> users, ArrayList<Document> documents){
    inflater = LayoutInflater.from(context);
    this.users = users;
    this.documents = documents;
    theContext = context;
    theActivity = (BaseActivity)context;
}

```

Another thing I can do with the adapter is to create a custom event for when a list item is selected. In my case I didn't want to have a view button at the bottom, but instead have it appear whenever a list item is selected. To do this an event listener can be added to each holder, when it is clicked it will become the selected item. The button will be set to visible and clicking the button will send the user to the document or profile page affiliated with what item is being held by the holder.

```

item.setOnClickListener((v) -> {
    ProfileViewHolder holder = (ProfileViewHolder)selectedHolder;
    if(selectedHolder != null) {
        holder.getUser().setSelected(false);
        holder.item.setBackgroundColor(0xFFFAFAFA);
        holder.btnView.setVisibility(btnView.INVISIBLE);
    }

    if(!ProfileViewHolder.this.equals(selectedHolder)){
        item.setBackgroundColor(0xFFCEC5D8);
        btnView.setVisibility(btnView.VISIBLE);
        ProfileViewHolder.this.getUser().setSelected(true);
        selectedHolder = ProfileViewHolder.this;
    } else {
        selectedHolder = null;
    }
});

```

To the home layout I added two of these recycler views and assigned an adapter to each of them, one for the list of observers and one for the documents associated with that user. I then requested the corresponding ArrayLists from the server and added them to each of the RecyclerView's. They're then wrapped in a ScrollLayout so that the list can be scrolled through by dragging the screen. The last thing I added to the home activity was series of buttons to access details, view their documents and view observer/observing users.

```

private void updateDocuments(){
    Response res = serverRequest(new Request(9));
    ArrayList<Document> documents = (ArrayList<Document>) res.getObject();

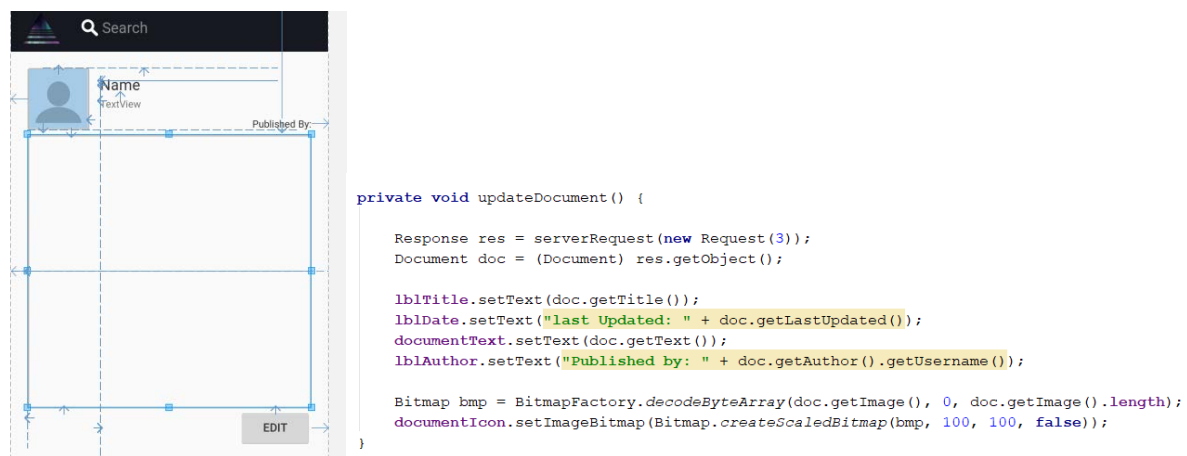
    adapter = new ItemAdapter(this, null, documents);
    documentRecycler.setAdapter(adapter);
    documentRecycler.setLayoutManager(new LinearLayoutManager(this));
    lblDocuments.setText("My Documents: (" + documents.size() + ")");
}

```

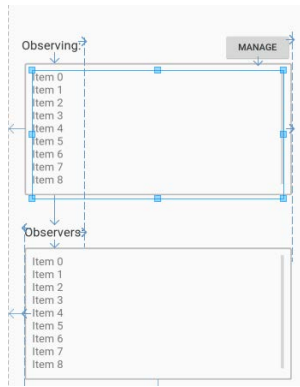
The detail and register activities act very similarly, they allow the user to input personal details about themselves into the provided EditText fields and then press save/register. This will obtain all the inputted data and place it into a new Profile object. This is then sent to the Register/Detail invoker via the socket within a Request object. From there the invoker will add the user details as it would before.



Similarly, the Document activity will simply obtain the document object from the document invoker and display the text, title and document image onto the layout elements. When the user presses the edit button the Text fields will become editable and the image will open the gallery browser, this is done by creating an ACTION_PICK intent and activating it with startActivityForResult() when the user closes the gallery it will return to the document activity with the result of what was chosen from the gallery. This can then be set as the ImageView's bitmap object. The document can then be saved in the same way as the user details but by forming a new document object instead and sending it back to the document invoker.



The observe activity will be used just to display the list of all profiles the user is observing and being observed by, this uses the same RecyclerView method found on the home activity a Remove button at the bottom of the page will get the selected user from the observing list and send a request to the server to remove that specified user from the OBSERVER_Table. The search activity works in the same way as the GUI, the query string is sent to search invoker which responds with a list of results, which is either profiles or documents depending which is selected by the spinner.

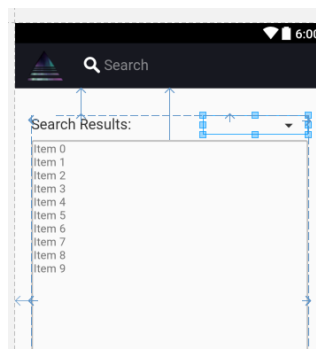


```
private void updateObserving() {
    ArrayList<Profile> observingList = (ArrayList<Profile>) serverRequest(new Request(4)).getObject();

    adapter = new ItemAdapter(this, observingList, null);
    observingRecyclerView.setAdapter(adapter);
    observingRecyclerView.setLayoutManager(new LinearLayoutManager(this));
    lblObserving.setText("Observing: (" + observingList.size() + ")");
}

private void updateObserver() {
    ArrayList<Profile> observerList = (ArrayList<Profile>) serverRequest(new Request(3)).getObject();

    adapter = new ItemAdapter(this, observerList, null);
    observerRecyclerView.setAdapter(adapter);
    observerRecyclerView.setLayoutManager(new LinearLayoutManager(this));
    lblObservers.setText("Observers: (" + observerList.size() + ")");
}
```

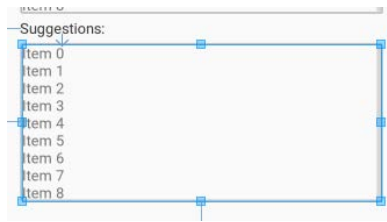


```
private void setResults(int pos) {
    ItemAdapter adapter = null;

    if (pos > 0) {
        Request req = new Request(5, searchBar.getText().toString());
        req.setValue(0);
        Response res = serverRequest(req);
        ArrayList<Document> results = (ArrayList<Document>) res.getObject();
        adapter = new ItemAdapter(this, null, results);
        lblSearch.setText("Search Results: (" + results.size() + ")");
    } else {
        ArrayList<Profile> results = (ArrayList<Profile>) serverRequest(new Request(4)).getObject();
        adapter = new ItemAdapter(this, results, null);
        lblSearch.setText("Search Results: (" + results.size() + ")");
    }

    searchRecyclerView.setAdapter(adapter);
    searchRecyclerView.setLayoutManager(new LinearLayoutManager(this));
}
```

The last thing to do is set up the suggestions. Suggestions will be a list of documents that the user can click on and view, like before it will be a populated RecyclerView. The list is obtained by sending the logged in user's profile ID, the invoker will then obtain the users unique VSM vector and compares the similarity to each documents in the system and returns the list of most similar documents by comparing each of the assigned similarity values to each other. I can now begin to test that the system creates the desired results.



```
private void updateSuggestions() {  
    Response res = serverRequest(new Request(10));  
    ArrayList<Document> documents = (ArrayList<Document>) res.getObject();  
  
    adapter = new ItemAdapter(this, null, documents);  
    suggestionRecycler.setAdapter(adapter);  
    suggestionRecycler.setLayoutManager(new LinearLayoutManager(this));  
}
```

Lastly, I had to test that my system would function over a WAN connection. To do this I created a port forwarding rule for the port that my server is running on. The app responded slower when changing between activities but this is expected as it is a slower connection. I could successfully view and share documents, add and manage users and receive suggestions that were relevant to the documents I was adding and viewing. Further testing will be required but I am satisfied with the functionality of my system.

Port forwarding

This function allows for incoming requests on specific port numbers to reach web servers, FTP servers and mail servers, etc:

Local IP	192 . 168 . 0 . <input type="text" value="11"/>
Local start port	<input type="text" value="1077"/>
Local end port	<input type="text" value="1077"/>
External start port	<input type="text" value="1077"/>
External end port	<input type="text" value="1077"/>
Protocol	<input type="text" value="TCP"/>
Enabled	<input type="text" value="On"/>

Testing

To measure the success of my implementation and to identify areas for improvement in my second iteration of development I will be performance end user testing. The testing will focus on usability, error tolerance, aesthetics, user satisfaction. I will use my requirements I used during planning to create the test.

Functional Testing

Requirements	Test	Expected Results	Status
<i>AF1, AF2, AF7, CF1, CF2, CF7</i>	A new unique user will register using the app, they will logout and then attempt login with incorrect then correct details.	<ul style="list-style-type: none"> - Registry complete, user logged in to home screen. - Logout is complete. - Login is unsuccessful (message: invalid username or password) - Login is successful, goes to home screen. 	Successful
<i>AF3, BF1</i>	User will make a new document, open the document and set a custom image, title and document text before pressing the save button, exit document then view it again.	<ul style="list-style-type: none"> - New document named "new post" appears in document list. - Clicking on "new post" opens a blank document. - (Message: "Changes Saved") when pressing save button. - Doc contains entered data when re-opened. 	Successful
<i>AF4, AF5</i>	The system containing a user with username "test" the user will enter "test" into the search bar and view the profile list. They will click on the first result and click "observe" on their profile. They will then go to their user page, locate "test" and press remove on their profile, then return to user page.	<ul style="list-style-type: none"> - Search results return one user of name "test". - Enters "test" profile page. - (Message: "Observing User") when click to observe. - User "test" is displayed in observing list. - (Message: "User removed") when click to remove. - User "test" no longer in observer list. 	Successful
<i>AF6</i>	Go to the users detail page, press on edit button, then enter new details along with a new profile image. Then press save, return to the home screen then enter detail page again.	<ul style="list-style-type: none"> - Edit page opened contains all previous information - (Message: "Changes Saved" once new information added and saved. - Edit page opened contains all new information. 	Successful
<i>AF8</i>	Attempt all of above tests with at minimum two devices open on different users simultaneously.	<ul style="list-style-type: none"> - All above tests return with positive results. 	Successful
<i>BF2, BF3, BF4, BF5, BF6, BF7, BF8</i>	Create a document containing the text "test", login to a new user and create a similar document with the same text. Return to the home screen.	<ul style="list-style-type: none"> - Suggestions list of new user contains the first document created. 	Successful
<i>BF8, BF9</i>	Following from the above test, the user must add another document containing the text "invalid" and then enter "test" into the search bar and change the spinner to "Documents".	<ul style="list-style-type: none"> - The search results only contain the documents containing the word "test" and not the document containing "invalid". 	Successful
<i>BF10</i>	The user must delete the newly added document containing the text "invalid", They must then complete the previous two tests.	<ul style="list-style-type: none"> - (Message: "Document Removed") when removing the document. - The previous two test return successful again. 	Successful
<i>CF3</i>	The user must navigate between the login, registration, home, all users, details, all documents, document and profile pages by clicking on the appropriate buttons.	<ul style="list-style-type: none"> - Each of the listed views are accessible by the user and the destination of each navigation button is the expected view the user anticipated. 	Successful

CF4	For all of the above listed pages, excluding the login and registration pages, the user must press the logout button provided in the action bar.	- Every time the logout button is pressed it sends the user to the login page and requests the user login again (ends the session).	Successful
CF5, CF6	With the database containing more than 10 users, the user must make a search for all users by entering nothing into the search bar and pressing enter. They must then attempt to view each resulting users.	- The search returns all of the 10 added users within the results box. - The box can be dragged down to view to users at the bottom of the list. - Clicking the view button will send the user to the corresponding user profile page.	Successful

Performance Testing

Requirements	Test	Expected Result	Status
AP1	Enter a valid user's login details. Starts at press login button, ends when user reaches home page. Try WAN and LAN connection.	Time < 5000ms	Unsuccessful (Time approx. = 7000ms)
AP2	Press the observe/remove button on a user's profile page. Ends when toast message appears and database is updated.	Time < 1000ms	Successful (Time approx. = 1000ms)
AP3	Take average time between saving a document, searching for a document/user and viewing a document/user.	Time < 5000ms (for each)	Successful (Time approx. = 2500ms)
CP1	Take average time switching between all views the user can access.	Time < 1000ms (for each)	Unsuccessful (Time approx. = 2000ms)
CP2	User updates their details, time between pressing save button and receiving "Changes Saved" toast message.	Time < 1000ms	Successful (Time approx. = 1000ms)
CP3	From the android OS main screen, the user should record the time between launching the application and viewing the login page.	Time < 10000ms	Successful (Time approx. = 3000ms)
CP4	User must attempt to launch and login to the system using an android device running Android API 15.	The system should behave normally in line with all above tests.	Successful

User Testing

The user testing was conducted by asking the user a series of questions and recording their response. The form that I used for user testing can be found attached to this report. I decided to test 8 users as this gave me variety of opinions, the success of the test will be a fraction of the users that answered yes or no.

Requirements	Question	Results
AU1	Were Documents suggested to you by the system relevant?	6/8 (Comment: There were no documents that were relevant)
AU2	Were search results accurate and relevant to your query?	8/8
AU3	Were you able to manage your profile data and delete it when you wanted to?	8/8
AU4	Were you able to connect to the users you wanted to and view their content?	7/8 (Comment: couldn't view observing documents from the home page)
AU5	Was it easy to view new/old posts made by other users?	6/8 (Comment: There is no way to order oldest to newest making it hard to find old posts)
BU1	Did suggestions reflect your changing interests overtime?	6/8 (Comment: Still lacked relevant documents at times)
CU1	Did the system help you to enter the correct data?	8/8
CU2	Were you able to view all the documents, users, and personal details that you wanted?	8/8
CU3	Were you able to add your own images in the way you wanted to?	7/8 (Comment: Images of a large size were not allowed in the system)
CU4	Did you know every time if an action was complete?	8/8
CU5	Was the application easy to use and navigate?	7/8 (Comment: button labelling a little ambiguous especially on home page)
CU6	Did the help page offer you the appropriate level of assistance?	8/8
CU7	Were all the sections, text fields and buttons labelled accurately?	7/8 (Comment: ambiguity with the buttons on the home page)
CU8	Were you able to navigate backwards effectively?	8/8
CU9	Was the logo distinctive enough to locate and recognize?	7/8 (Comment: logo possibly too small on the action bar)
CU10	Did you find the application felt professional?	8/8

Summary

Evaluation

After Testing all of my initial requirements I have a better idea of the success of my project. Although it is not certain that my system is void of errors; From my functional testing I can see that the app did not suffer any fatal errors when carrying out the basic use cases and also completed them successfully with the correct results, this is important knowledge as when it comes to my second iteration of development, I can focus on improvements rather than fixing old features.

However, my testing did show that the system did not perform as well as I would have liked. What it struggled with was logging in as fast as it should and changing between views on the application. As for the user testing it seems a majority of requirements were satisfied, however there could be more design improvements to the button labelling as well as the recommender system which did not always return relevant results. This is likely however down to a lack of documents in the system that had any similarity to the documents the user was comparing to.

Reflection

When I first started the project, I began implementing the vector space model first since I expected it to be the hardest of the three phases. It did turn out to be the most difficult part but only due to an oversight with how the social platform was designed, since it needed to update the database continuously, the VSM couldn't operate independently. I had to go back and redesign the VSM to work flexibly with the database. If I were to start the project again I would spend more time planning to ensure that each component would work together as intended.

The results I got back from my testing suggest the performance of my system are not reaching the requirements. However, this is only the case when the application is operating over a WAN connection as it takes longer to upload and download the data. When planning my testing in the future I will be sure to consider the connection speed having an effect on the system performance not just the CPU time.

I have learnt a lot about handling large amounts of data, even with the system I have created just a handful of users can generate a vast amount of information that can be used to improve the recommender system. It acts as a perfect microcosm for how the largest tech companies in the world manage to provide such complex social services.

As members of the tech industry we should always aim to provide the best data transparency and security for users as is possible. To build trust and continue to improve people's quality of life through advances in technology.

Overall, I am very satisfied with what I have been able to create. It has been a difficult task without a lot of prior knowledge related to information retrieval and android application development.

However, due to my research, planning and the programming skills I have learnt in the past 3 years, I was able to create a functioning social platform that I am excited to reveal at the degree show.

Future Improvement

If I were to begin a second development cycle of my project I would first improve the aspects that I discovered in my performance and user testing. To do this I would first integrate some rating system for all the documents and possibly users, as well as a comment sections. That information could be used to improve the relevance of the suggestions produced by the recommender system. Once that is implemented I would consider the following:

- Redesign the layouts to contain bigger and more intuitive buttons so that the user can navigate more effectively.
- Add tabs to the home page so that users can view recent posts made by other users easily.
- Give the user more option with how documents are ordered, such as ordering by oldest to newest.
- Implement loading screens to signal to the user that the app is working on a request and not frozen.
- Implement 2-way encryption on user data flowing to and from each client.
- Implement stronger login systems such as 2-factor authentication to prevent security breaches.
- Create a backup system and routine to manage the data more appropriately and ensure user information is never lost.
- Implement a form of direct messaging between users so that they can communicate their thoughts and opinions without sharing them publicly.
- Extend the recommender system to including profile suggestions as well, so that users can find other users similar to them based on their content.

References

1. Introduction to information retrieval

- **In-text:** (Manning, Raghavan and Schütze, 2008)
- **Bibliography:** Manning, C., Raghavan, P. and Schütze, H. (2008). *Introduction to information retrieval*. Cambridge: Cambridge University Press.
- **Annotation:** This source focuses heavily on user feedback, complex queries and clustering. It described the algorithms and mathematic proof behind information retrieval from a large data set. The book was exceptionally valuable in my research as filtering and extracting text data from my systems database will be a big part of my project. It is very detailed and provides great explanations of equations and diagrams for data clustering.

2. Web data mining

- **In-text:** (Liu, 2008)
- **Bibliography:** Liu, B. (2008). *Web data mining*. Berlin: Springer.
- **Annotation:** A great book for web based data collection, it detailed the process of implementing a web crawler, data analysis and supervised learning to form accurate information clustering. As well as an outline of user profiling and how it is utilized throughout the web. Very helpful source for my research, especially for when I will be creating my recommender system. Contained less information on clustering, however far more detail on how the internet is used for information retrieval.

3. Algorithms of the Intelligent Web

- **In-text:** (Marmanis and Babenko, 2011)
- **Bibliography:** Marmanis, H. and Babenko, D. (2011). *Algorithms of the intelligent Web*. Greenwich, Conn.: Manning.
- **Annotation:** This source focuses on data collection and filtering as well - however explores machine learning and the complex algorithms that are used in AI systems. This research will be similarly beneficial to building my recommender system. It also covers a lot of information related to object oriented design; this is very helpful since I will be building a majority of my application in java. The book also benefits from being the most recent of my sources, including more up-to-date methods and examples.

4. Data is the New Oil

- **In-text:** (Spotlessdata.com, 2018)
- **Bibliography:** Spotlessdata.com. (2018). *Data is the new oil*. [online] Available at: <https://spotlessdata.com/blog/data-new-oil> [Accessed 2 May 2018].
- **Annotation:** Provides information on data collection and storage by big companies.

5. EU GDPR Information Portal

- **In-text:** (EU GDPR Portal, 2018)
- **Bibliography:** EU GDPR Portal. (2018). *EU GDPR Information Portal*. [online] Available at: <https://www.eugdpr.org> [Accessed 2 May 2018].
- **Annotation:** Provides information on the new GDPR regulations put in places as of 2018.

6. Statista - The Statics Portal for Market Data.

- **In-text:** (Statista.com, 2018)
- **Bibliography:** Statista.com. (2018). • *Statista - The Statistics Portal for Market Data, Market Research and Market Studies*. [online] Available at: <https://www.statista.com/> [Accessed 4 May 2018].
- **Annotation:** Provides quantitative data on company financial and technical statistics.

7. Jacquard Machine Analyzed and Explained

- **In-text:** (Lhldigital.lindahall.org, 2018)
- **Bibliography:** Lhldigital.lindahall.org. (2018). *Jacquard machine analyzed and explained. :: History of Engineering & Technology*. [online] Available at: http://lhldigital.lindahall.org/cdm/ref/collection/eng_tech/id/3983 [Accessed 5 May 2018].
- **Annotation:** Provides information on the jacquard machine.

8. Domesday History Magazine

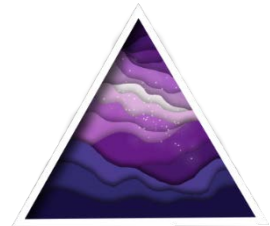
- **In-text:** (History-magazine.com, 2018)
- **Bibliography:** History-magazine.com. (2018). *History Magazine*. [online] Available at: <http://www.history-magazine.com/domesday.html> [Accessed 2 May 2018].
- **Annotation:** Provides information on the Domesday book.

Appendix

Aurora – User Guide and Demonstration

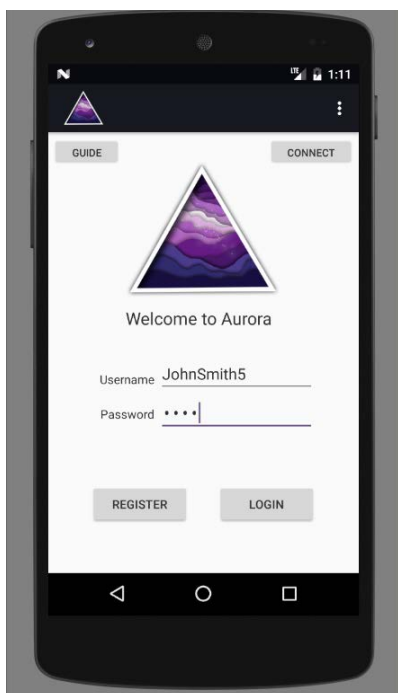
Introduction

Aurora is a social sharing and recommendation system. It allows users to create text-based content and share it with other users. It also profiles users to produce content suggestions relevant to the user's recent activity and interests. The following document will showcase the applications features and provide instructions for efficient use.



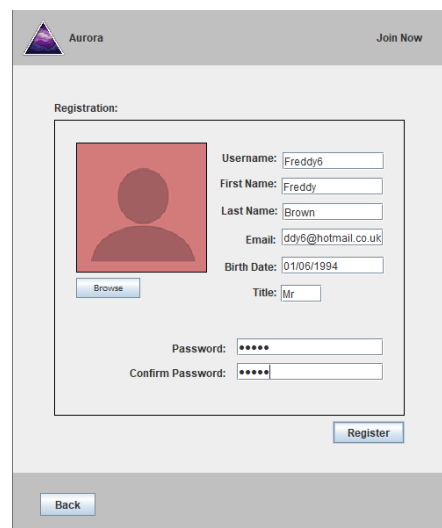
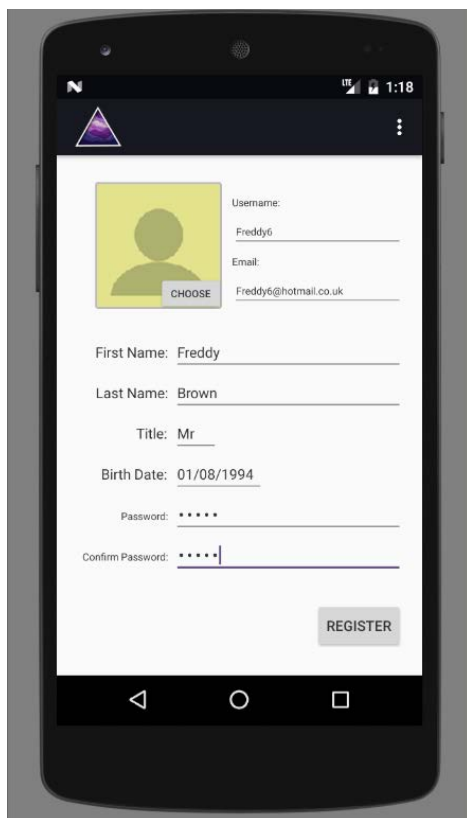
Login Screen

The login screen is the first page they will be sent to once they have launched the application. From here the user can either enter their unique username and password and press the “Login” button to log in to the system. If they have not already created an account they can press the “Register” button sending them to the registration form. If the user is unsure of how to use the system, then they can press on the “Guide” button which will send them to the help page.



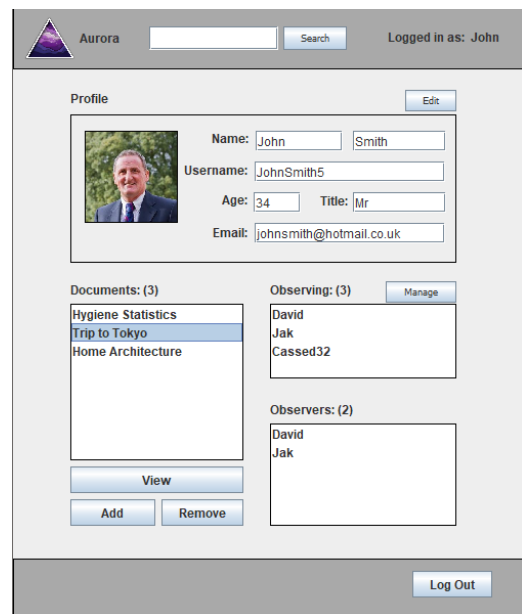
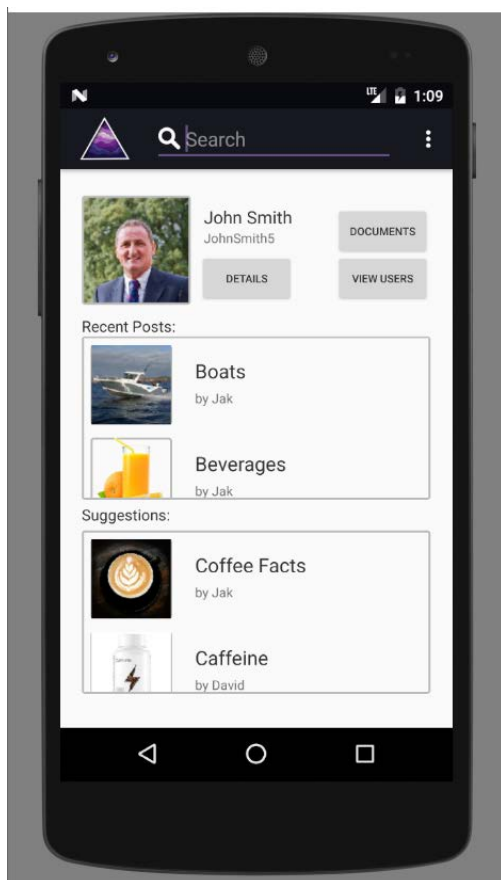
Registration

Registration can be completed from the registration form, accessed from the login page. It allows users to create a new account. To do this the user will enter in a unique username, as well as all their personal details. Each user will be assigned a colour by default, this will be displayed in their profile picture at the top of the page. If the user wants a custom image they can choose one by tapping or clicking on the image. This will open their gallery or file directory in which they can choose the image they desire. They will also need to create a password which they enter into both the “password” and “confirm password” fields. Once all the information is entered the user will be able to press the “Register” button. If all fields have been entered correctly the user will be logged in and sent to their unique home screen. If not, then the user will be informed that one or more of the fields are invalid.



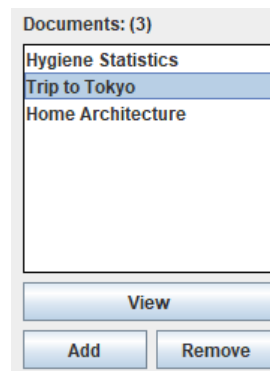
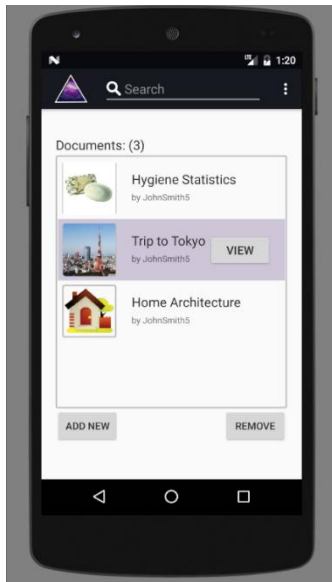
Home Screen

The home screen is the main view of the application. From here the user can press the “Edit Details” button to view and change their personal information. The “Users” button will send them to a page they can view all profiles they are observing and being observed by. The “Documents” button sends them to a page they can view all the documents they have created. This page also features a list of documents recently posted by profiles the user is observing and some documents that are suggested to the user based on their previous activity.



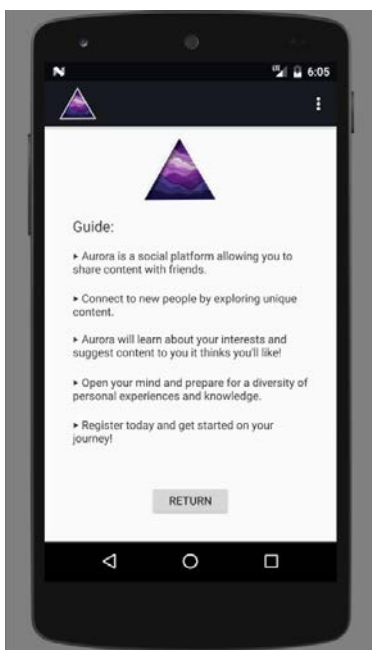
Manage Documents

To manage documents the user has created they can press on the “Documents” button on their home screen. They will be sent to a page containing all documents they have created. From here they can select the document they wish to interact with. The document will become highlighted, they can then either press the “View” button to view that specific document, or press the “Delete” button to permanently remove it from the system.



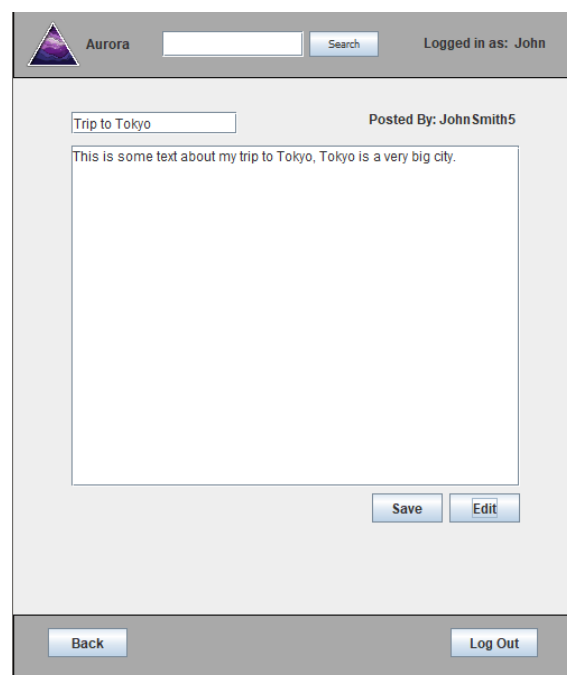
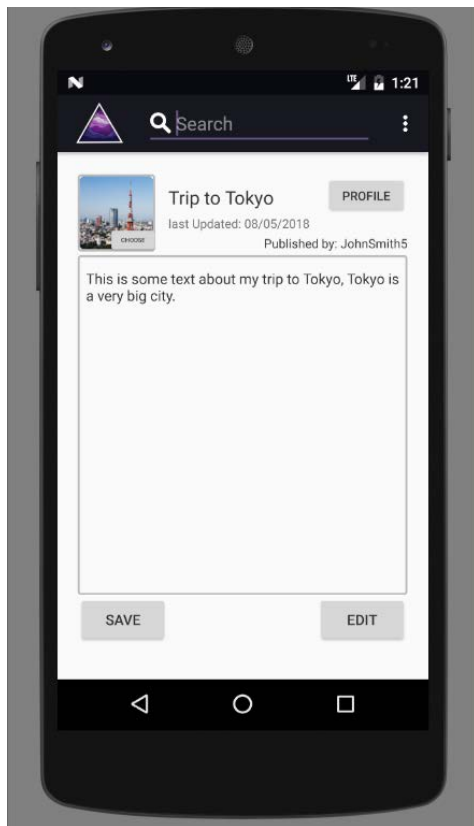
Guide

Users can also access a short explanation of what the system is, and how it should be used by clicking on the “Guide” button on the login screen.



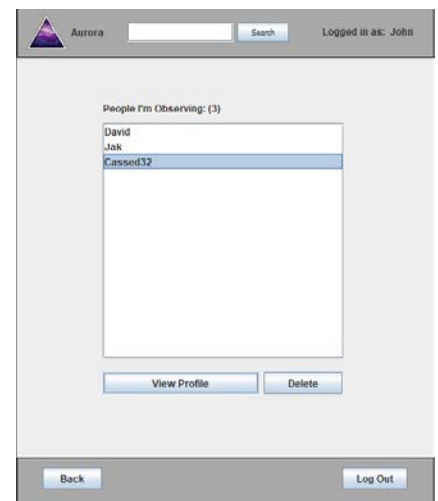
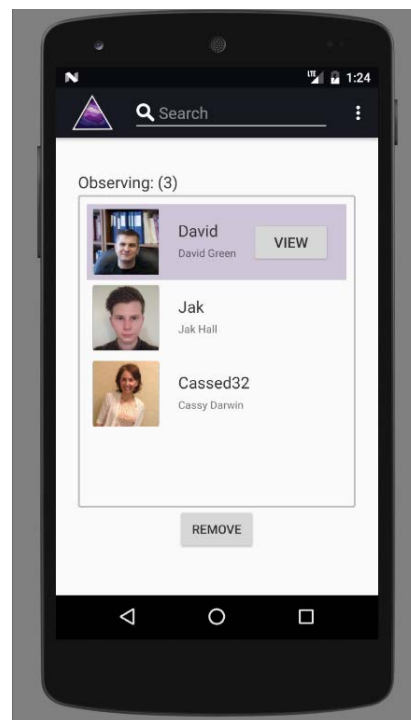
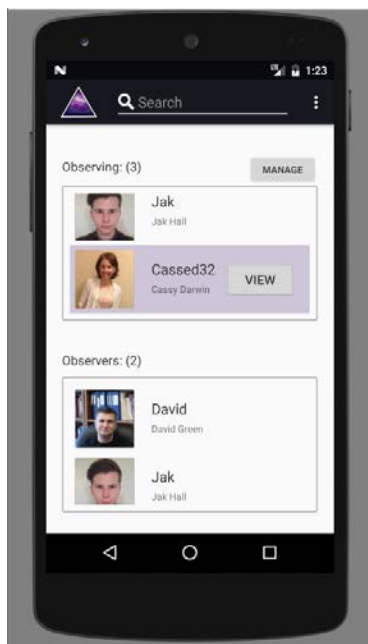
Editing Documents

When viewing a document that the user has personally created, an “Edit” button will be displayed at the bottom of the page. When they press this the page will become editable and a “Save” button will appear. They can now change the title and document text by clicking/tapping on the corresponding areas, as well as change the document image by pressing on the current image in the top left corner, this will send the user to their gallery/file browser allowing them to choose the image they desire. Once the user is finished editing the page, they can press the “Save” button to commit any changes to the system. When pressed the user will be notified and the document will change back to being un-editable.



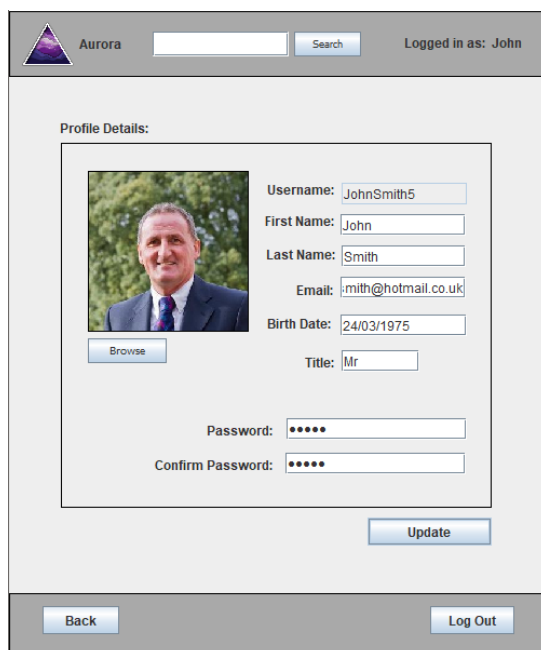
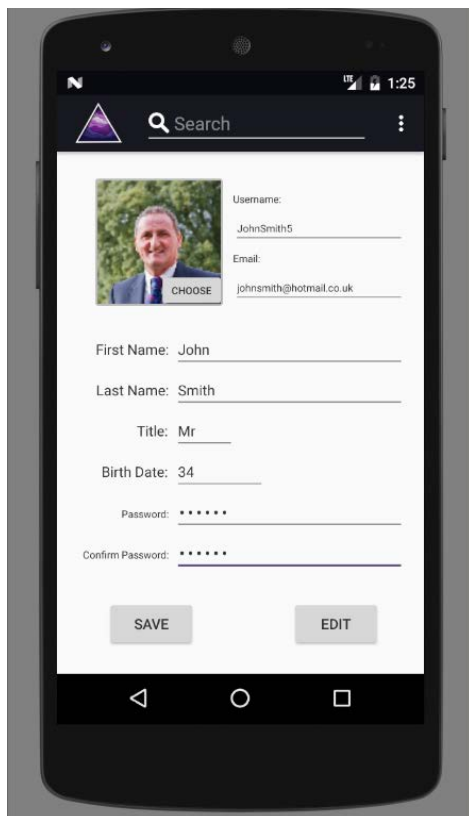
Manage Users

From the home screen the user can press the “Users” button to access a page which has two lists. The first list is all profiles the user is observing, the second list is all the profiles observing the user. The user can then select a user by tapping on their image or name, it will become highlighted. They can then press the “View” button to go that users profile page. To remove a profile from the users observing list, on desktop they can simply press the “Delete” button. On mobile, the user can press the “Manage” button at the top of the page to access a new page containing only the observing list. From their they can select the profile as before and press the “Delete” button at the bottom of the page. The link between the users will now be removed from the system.



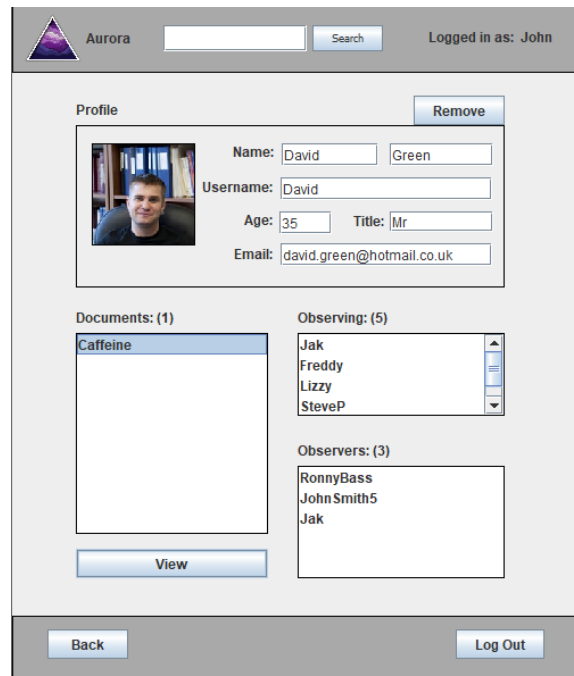
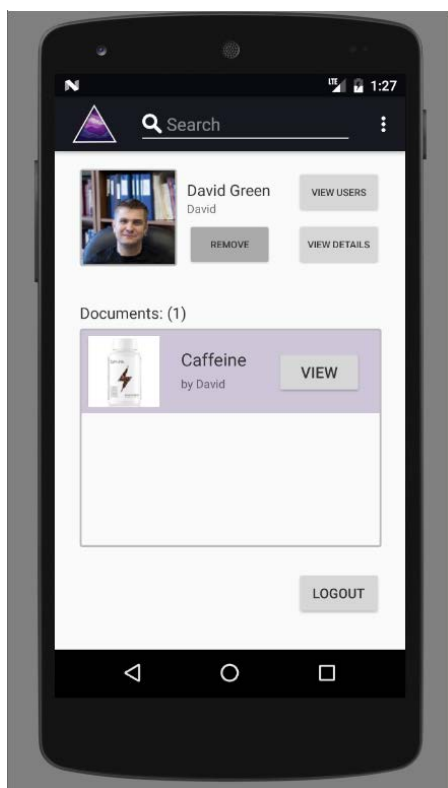
Manage Details

The user can change personal information stored about them at any time. To do this they can press the “Details” button on the home page. This will send them to a new page containing all the details the system is storing about them. When they press the “Edit” button the field will become editable and they will be able to change their details. By pressing on the profile image they can access their gallery/file explorer to choose a new image they desire. To keep the password the same, the user can leave the password field empty. Otherwise they will need to enter a new password into both the “Password” and “Confirm Password” fields. When the user is done making changes they can press the “Save” button to commit changes to the system provided all information they entered was valid. The username, however, is unique and cannot be changed. To have a new username the user must create a new account.



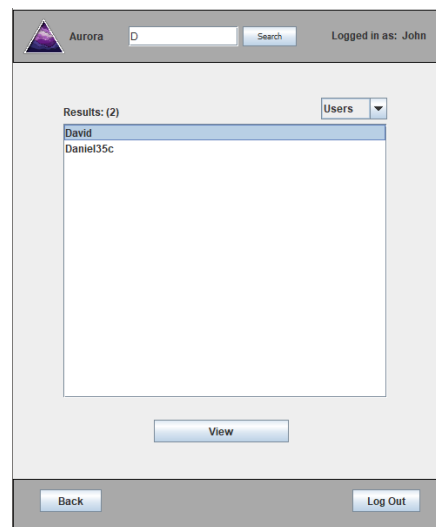
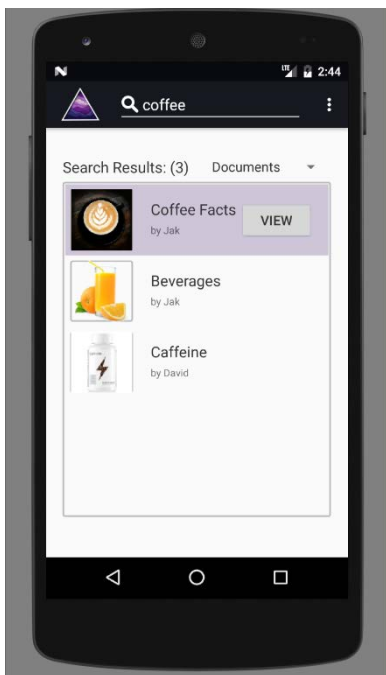
Viewing Other Users

Along with view their own documents and details a user can view another profile by selecting the user and pressing the “View” button. This will send them to the selected users profile page. Additionally, profiles can be accessed by pressing the “Profile” button on any document which will send the user to the authors page. From the profile they will have access to familiar navigation to the home screen, however the “Details”, “Documents” and “Users” buttons will send the user to pages which show the viewed users information instead. This is a good way to find new documents and related users. The user will not be able to edit these pages however as they are managed only by the profiles user.



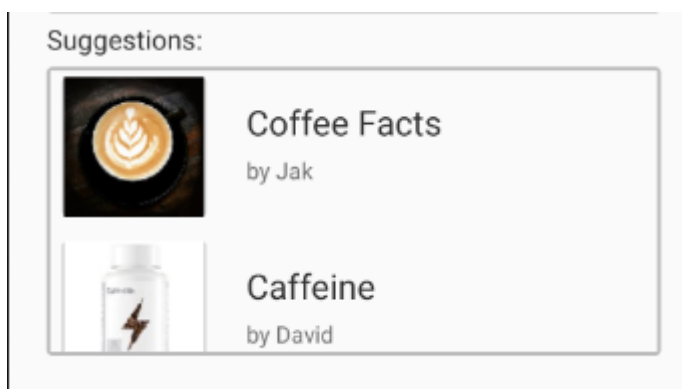
Search Bar

A good way to explore the users and content available is to use the search bar, by inputting some text and pressing enter, the user will be sent to the search screen and be able to view all relevant results. For documents, complex queries can be made to search for documents of highest similarity to the query. The dropdown list on the right-hand side allow the user to choose whether they want to search for other users or content.



Suggestions

Suggestions are a helpful tool in allowing the user to explore new documents and connect to new users. The suggestions are unique to each user and are affected by the users previous activity, such as view, adding and deleting documents. The suggestions will dynamically present documents to the user which have the highest similarity to other documents they have interacted with. The suggestions can be accessed from the bottom of the users home screen.

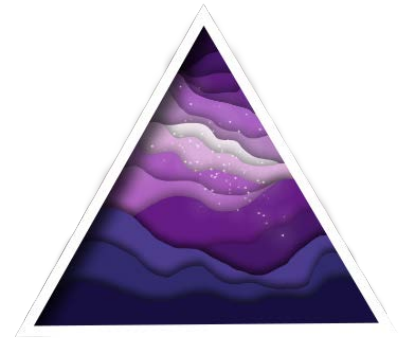


Aurora – User Testing

Introduction

As part of my final university project I have built a social platform allowing users to share and explore user created content. To assess the success of my project and identify potential areas for improvement I need your opinion. Please take a short time to test the program I have created, and then answer the following questions. For each question please circle or highlight Yes or No, also please provide a comment if applicable. Thank you in advance for your time.

Jak Hall.



User Assessment

Question 1

Were Documents suggested to you by the system relevant?

Yes / No

Comment: _____

Question 2

Were search results accurate and relevant to your query?

Yes / No

Comment: _____

Question 3

Were you able to manage your profile data and delete it when you wanted to?

Yes / No

Comment: _____

Question 4

Were you able to connect to the users you wanted to and view their content?

Yes / No

Comment: _____

Question 5

Was it easy to view new/old posts made by other users?

Yes / No

Comment: _____

Question 6

Did suggestions reflect your changing interests overtime?

Yes / No

Comment: _____

Question 7

Did the system help you to enter the correct data?

Yes / No

Comment: _____

Question 8

Were you able to view all the documents, users, and personal details that you wanted?

Yes / No

Comment: _____

Question 9

Were you able to add your own images in the way you wanted to?

Yes / No

Comment: _____

Question 10

Did you know every time if an action was complete?

Yes / No

Comment: _____

Question 11

Was the application easy to use and navigate?

Yes / No

Comment: _____

Question 12

Did the help page offer you the appropriate level of assistance?

Yes / No

Comment: _____

Question 13

Were all the sections, text fields and buttons labelled accurately?

Yes / No

Comment: _____

Question 14

Were you able to navigate backwards effectively?

Yes / No

Comment: _____

Question 15

Was the logo distinctive enough to locate and recognize?

Yes / No

Comment: _____

Question 16

Did you find the application felt professional?

Yes / No

Comment: _____

Consent

- ☐ Please tick this box to confirm that you have read all questions and answered honestly.
- ☐ Please tick this box to confirm that you are aware that the information you have provided will be used internally for evaluation and further analysis. (No personal information will be collected or used.)

Signature: _____

Aurora – Project Log

Introduction

To maintain consistent progress and evidence of my project development I have kept a project log. The log has the date the entry was made, the guideline goals I planned to achieve by that date about my planned scheduling and a reflection on what I actually achieved and any difficulties I faced.

Log Table

Date	Goals	Reflection
06/01/2018	<ul style="list-style-type: none"> - Investigate TFIDF. - Create test documents. - Prototype Parsing text. 	After doing a lot of research previously, I didn't need too much more information to understand how I'd begin building the system. I spent most of my time working out how to get the JVM to read the text files, it was a challenge at first as certain characters would end the string. However, I eventually got it working.
13/01/2018	<ul style="list-style-type: none"> - Implement term frequency assigning. - Implement document frequency assigning. 	What I realized is that the stemming will need to be completed before like terms can be grouped. Instead of working on document frequency I began implementing the porter stemming.
20/01/2018	<ul style="list-style-type: none"> - Investigate Porter stemming. - Begin implementing stemming into project. 	I could finish the porter stemming and the term frequency now works as it should. I then began to create a way to find all documents with certain terms in them, however this will be easier later once I have implemented a database.
27/01/2018	<ul style="list-style-type: none"> -Finish implementing stemming. -Implement common word removal. - Begin TFIDF assigning. 	I had already finished stemming so I spent the week working on the TFIDF functionality. This turned out more difficult than I thought since every time a document is added or removed all documents sharing like terms are also affected.
04/02/2018	<ul style="list-style-type: none"> - Finish TFIDF assigning. - Implement vector space model. - Work on converting document objects to vectors. 	I was able to finish the TFIDF calculation, which made it easy to form vector objects by assigning the TFIDF as the vector weighting. The vector space model is working however it takes a long time to add each document.
11/02/2018	<ul style="list-style-type: none"> - Investigate and implement cosine similarity. - Build way to view similarity between documents. 	I found it very challenging to implement the cosine similarity, I was getting seemingly random results when comparing documents. I soon realized however this was because my

		vector space model was storing terms in inconsistent order, this meant like terms were being compared incorrectly. I also was able to speed up the time for documents to be added and removed.
18/02/2018	- Create database. - Connect database to system.	I did some research and decided to use an SQLite database to store my data, I found it easy to connect it to the java system.
25/02/2018	- Implement method for user adding/removing editing. - Begin document management.	Beginning to create the social platform was hard and I soon realized I needed more tables in my database, I was able to get a simple way to add and remove users however it will be easier to test with a GUI.
02/03/2018	- Finish document management. - Add method for query and profile vectors.	Adding and removing documents turned out to be very similar to the management of users, so I was able to implement this very easily. I had to create additional tables in my database to handle adding profiles vectors to the system.
09/03/2018	- Begin on desktop GUI - Implement Registration and Search functionality.	I decided to prototype a GUI earlier in the month because I needed a way to test the system. If I were to start again I would have created the desktop application sooner. The register and search functions were very similar to methods I had already used for editing user details and the query vectors so It was quick for me to implement.
16/03/2018	- Finish desktop GUI	The GUI was already finished by this week. I decided to spend some time to refactor my code, add additional comments and test some of the validation checks.
23/03/2018	- Create server application for mobile to connect to.	I initially planned to remote method invocation for creating a connection between the mobile and desktop system, however android does not support this functionality. Instead I decided to build end to end sockets which will allow the two systems to communicate.
30/03/2018	- Begin working on mobile application.	I am less familiar with android development so I was slow to build the application. I was able to create the client-side socket however and send basic byte information between the programs on a LAN connection.
07/04/2018	- Continue to work on mobile application.	Having struggled the first week, I was much quicker to implement the application as I was more familiar with the sdk. The views of the application were very similar to the views of the GUI so I was able to reuse some of the code.

14/04/2018	- Finish mobile application.	The mobile application was mostly complete so I went back to implement features to the desktop GUI that were not yet present. I also tested some concurrency by using both systems together.
21/04/2018	- Implement recommender system into desktop GUI and mobile application.	Because I build the VSM before I implemented the database and server, it was very hard to get it to perform efficiently. I decided to spend some time to redesign the VSM so that it would work fluidly with the database by adding a few more tables and refactoring my code. I was then able to easily retrieve similar documents from the system and display them to the user.
28/04/2018	- Conduct system and user testing. - Evaluation of project.	A majority of my project is complete so I was able to focus on testing, I created some functional and performance tests based upon my system requirements and conducted some user testing on select individuals to evaluate the success of the system.
05/05/2018	- Finish documentation and user guide.	I now just needed to format screen shots and implement some instructions on using the system. I also included a guide page on my applications so that users can inform themselves on how to utilize the system.