ICS 2205 - Project
Web Intelligence

Daniel Magro, Luke Ellul and
Stephanie Chetcuti

An analysis of Hillary Clinton’s Emails

D2: Text Analysis

i) Extracting the Text from each Email

The Database of Emails[D2-1] was provided as an SQLite Database. Thus, in order to parse the data set, first an **Email (Email.java)** class was constructed, which defines the set of attributes each email should have when parsed, and also a set of methods applicable to each Email. Each Email that is parsed will be an instance of the Email.java class, and will thus store the Recipient (String - to), Sender (String - from) and the body of the email, i.e. the Subject and the Body (ArrayList of Strings - text). A Class **EmailReader**[D2-2] was created which first connects to the SQLite database using the JDBC driver[D2-3]. After the connection to the database is made, the database is queried with "`SELECT MetadataTo, MetadataFrom, ExtractedSubject, ExtractedBodyText FROM Emails`". A loop goes through every tuple of the database, and assigns the data extracted from the current tuple to a variable respectively, i.e. **to** = MetadataTo, **from** = MetadataFrom, and each element of the **text** ArrayList is set to each word inside the ExtractedSubject and ExtractedBodyText. When all variables have been initialised, a new email object is created, initialised with the to, from and text variables, and added to the ArrayList of Emails. When all the tuples inside the database have been read, the ArrayList of Emails is returned to the main method.

ii) Performing Lexical Analysis - Word Separation + Case Folding

The words inside the Subject and Body of each email are already separated by all whitespaces and stored as elements of type String inside an ArrayList named text (each email has its own text arraylist). This was done when reading the database of emails inside EmailReader, with the following code:

```
String emailText = rs.getString("ExtractedSubject") + " " +
rs.getString("ExtractedBodyText");
ArrayList<String> text = new ArrayList<>(Arrays.asList(emailText.split("\\s")));
```

Then the method **foldCase** is called and the collection (arraylist) of emails is passed as a parameter. The foldCase method loops through every email. For each email,it loops through every element of the text array, and converts the text to lowercase.

iii) Stop-Word Removal

The list of stopwords[D2-4] is stored inside **stopwords.txt**. The removeStopWords method is called and the ArrayList of Emails is passed as an argument. The method first reads every word inside stopwords.txt and stores it inside an array. It then iterates through every word inside every email, and if a word is a stop word, it is removed from the ArrayList of text.

iv) Reducing Terms to their Stems - Porter's Stemmer

An implementation for Porter's Stemmer was found online[D2-5] and saved inside Stemmer.java. A **stemText** method was created which applies the implementation found online to every word. The stemText method iterates through every word in every email. For each word, this method first creates a new instance of the Stemmer class, adds each letter of the word one by one, then calls the stem function of the Stemmer and finally the stemmer's toString value is stored instead of the word.

v) Calculating the Term Weights using TF-IDF and converting Emails between 2 Correspondents to Documents.

Firstly, the Emails had to be converted into Documents.
To do this, a **Document** class was created, which defines the attributes each Document should have, these being:
String personA, String personB, ArrayList<String> text, ArrayList<**Keyword**> highestNPercentTerms.
The 2 Strings store the names of the 2 correspondents of the email thread. The ArrayList of Strings named text will store all the words exchanged between 2 correspondents. The ArrayList of Keywords will store the highest n% of terms, along with their weighting in the document.
An ArrayList of Documents was declared, and was initialised by the **createDocuments** method, which takes the List of Emails as a parameter. This method iterates through all the emails, and checks whether a document for the interactions between those 2 correspondents exists. If a document already exists, the text in the email is simply added to the document's text ArrayList. If not, a new document is created, initialised with the correspondents' email addresses and text, and added to the ArrayList of Documents. When all the emails have been processed, the ArrayList of Documents is returned to the main method.
Now that the List of Documents has been compiled, the **calculateTermWeight** method is called, and the List of Documents is passed as an argument.
First, the method calls the **findUniqueTerms** method, which goes through all the words in every document, and if it is a unique term (i.e. term is not already in the list), adds it to the list of uniqueTerms.
Once the unique terms have been found, a termByDocFreqMatrix (Term x Document Frequency Matrix) is declared, of as many rows as there are uniqueTerms (49,577) and as many columns as there are documents (582). Then, each cell is filled using the **calculateFrequency** method, which takes the term and the document as parameters.
Next, an array **docFreq** of equal length to the uniqueTerms array is filled with however many documents each word appears in using the **findDocFreqs** method.
Lastly, the termByDocWeightMatrix is initialised using the formula
$weight[i][j] = termFrequency[i][j] * log(Number\ of\ docs\ /\ docFreq[i])$.

An alternate formula is programmed, one where the termFrequency is divided by the total number of terms in the document, this was however commented out and replaced by the implementation above for more sensible weightings.

vi) Building a Keyword-Cloud for each Document

Now that the term weights for each term in each document have been calculated, the highest weighted n% of terms can be chosen to form part of the word cloud for each document.
The **findHighestNpercent** method takes the ArrayList of Documents and the termByDocWeightMatrix as arguments.
For each document, the method first creates a list of non-zero term weights. Then it sorts the term weights in ascending order, and chooses the cut-off value as the (0.9 * number of non-zero term weights)th element of the wordWeights array (0.9 for top 10% of words). Then, all the terms with a weight higher than the cut off are added to the ArrayList of KeyWords, highestNPercentTerms, of each document, along with their weight.

After all the keywords are found, the documents are written to disk in a .JSON format using the **writeDocsToJSON** method inside the **JSONWriter** class. One example extracted from the output is the following:

{"highestNPercentTerms":[{"weight":288.3492271129372,"keyword":"trump"},{"weight":112.3492272,"keyword":"afghan"}],"personB":"H","personA":"stallbott","text":["follow","afghan","matter","jake","sullivan","(dos)","trump","america","releas","part"]}

This JSON file was essential so that the Visualisations done in D3 could retrieve the data they require from this file.

The code for the JSON file writer was adapted from [D2-6]. The jar for the JSON functions was obtained from [D2-7].

vi) Building a Keyword-Cloud for each Correspondent

For this, the **findHighestNpercentPerCorrespondent** method was created, along with a **Correspondent** class.
This Correspondent class is very similar to the Documents class, however only stores one email address, rather than two.
The method first traverses all the emails again. For each email, it first checks whether the Sender (‘from’ email address) is already in the list of correspondents. If it is, the text of that email is added to the text of that correspondent, if not, a new correspondent is added to the list, along with the text of that email.

-Due to time constraints, not all of the following was implemented, however it is explained how it would have been implemented.-

A Term x ‘Correspondent’ Weight matrix would have been constructed for the list of correspondents, using the same method used for the Term By Document Weight Matrix.

Then, the highestNPercent of Terms per correspondent would have been found, again using the same method used for the documents.

Then, the **writeCorrespondentsToJSON** method is called, which in its current implementation simply creates a JSON file with entries for all the correspondents. This was required for the Visualisations constructed in D3. If the keywords were found, they would also be written to the JSON file, as they were for the documents.