Hurricane Irma

Team 9:
Raja Venkata Satya Phanindra Chava
Siddharth Dhar
Yamini Gaur
Pranavi Rambhakta
Sourabh Shetty

Instructor:
Dr. Edward A. Fox

December 13, 2018
## Contents

List of Figures .......................... 3
List of Tables ............................ 4

1 Abstract ................................ 5

2 Acknowledgement ......................... 7

3 Introduction ............................ 8
  3.1 Background ........................... 8
  3.2 Framework ............................ 9

4 Literature Review ....................... 10
  4.1 NLTK ................................. 10
  4.2 Gensim ............................... 10
  4.3 Classification Techniques .......... 10
  4.4 Deep Learning Techniques .......... 11

5 Data Preprocessing ...................... 12
  5.1 Noise Removal ....................... 14
  5.2 Normalization ....................... 16
  5.3 Tokenization ......................... 16

6 Data Exploration ....................... 17
  6.1 Most Frequent Words ............... 17
    6.1.1 Lemmatization .................... 19
    6.1.2 POS Tagging ...................... 19
  6.2 Bigrams .............................. 21
  6.3 Named Entity Extraction .......... 22
  6.4 LDA Topic Modeling ................. 23

7 Classification .......................... 25
  7.1 Mahout Classifier .................... 25
  7.2 Decision Rules Classifier .......... 27

8 Clustering .............................. 29

9 Abstractive Summaries .................. 31
  9.1 About the Model ...................... 32
  9.2 Preprocessing the Data ............. 33
  9.3 Training the Model ................... 35
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Framework for approaching textual data [21].</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>High-level view [22] of the iterative data preprocessing model.</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>Text data preprocessing framework [21].</td>
<td>14</td>
</tr>
<tr>
<td>4</td>
<td>Code Snippet for jusText HTML cleaning.</td>
<td>15</td>
</tr>
<tr>
<td>5</td>
<td>Code Snippet for tokenization.</td>
<td>16</td>
</tr>
<tr>
<td>6</td>
<td>Most frequent words after tokenization.</td>
<td>17</td>
</tr>
<tr>
<td>7</td>
<td>NLTK stopwords.</td>
<td>18</td>
</tr>
<tr>
<td>8</td>
<td>Most frequent words after stop word removal.</td>
<td>18</td>
</tr>
<tr>
<td>9</td>
<td>Code Snippet for Lemmatization using POS tagging.</td>
<td>20</td>
</tr>
<tr>
<td>10</td>
<td>Most frequent words after lemmatization using POS tags.</td>
<td>20</td>
</tr>
<tr>
<td>11</td>
<td>Bar graph of the most frequent words.</td>
<td>21</td>
</tr>
<tr>
<td>12</td>
<td>Most frequent words.</td>
<td>21</td>
</tr>
<tr>
<td>13</td>
<td>Code Snippet for generating bigrams.</td>
<td>22</td>
</tr>
<tr>
<td>14</td>
<td>List of bigrams and count of occurrences in the corpus.</td>
<td>22</td>
</tr>
<tr>
<td>15</td>
<td>Named entities.</td>
<td>23</td>
</tr>
<tr>
<td>16</td>
<td>Code Snippet for LDA Topic Modeling.</td>
<td>23</td>
</tr>
<tr>
<td>17</td>
<td>Topics generated by LDA topic modeling.</td>
<td>24</td>
</tr>
<tr>
<td>18</td>
<td>Code Snippet for duplicate removal.</td>
<td>25</td>
</tr>
<tr>
<td>19</td>
<td>Code Snippet for Classification using Mahout.</td>
<td>26</td>
</tr>
<tr>
<td>20</td>
<td>Classification results.</td>
<td>27</td>
</tr>
<tr>
<td>21</td>
<td>Code Snippet for Decision Rule Classification.</td>
<td>28</td>
</tr>
<tr>
<td>22</td>
<td>Code Snippet for Clustering using Mahout.</td>
<td>29</td>
</tr>
<tr>
<td>23</td>
<td>Relevant clustered documents.</td>
<td>30</td>
</tr>
<tr>
<td>24</td>
<td>Irrelevant clustered documents.</td>
<td>30</td>
</tr>
<tr>
<td>25</td>
<td>Sequence to sequence model. [18]</td>
<td>31</td>
</tr>
<tr>
<td>26</td>
<td>Pointer-Generator model. [18]</td>
<td>32</td>
</tr>
<tr>
<td>27</td>
<td>Code snippet to get .story files from the input JSON file.</td>
<td>34</td>
</tr>
<tr>
<td>28</td>
<td>Output snippet when generating binary files and vocab files from the .story files</td>
<td>34</td>
</tr>
<tr>
<td>29</td>
<td>Solr Database</td>
<td>43</td>
</tr>
<tr>
<td>30</td>
<td>Querying the Solr database</td>
<td>44</td>
</tr>
<tr>
<td>31</td>
<td>Running the ArchiveSpark scala script.</td>
<td>49</td>
</tr>
</tbody>
</table>
List of Tables

1  Results after noise removal using jusText ............... 15
1 Abstract

With the increased rate of content generation on the Internet, there is a pressing need for making tools to automate the process of extracting meaningful data. Big data analytics deals with researching patterns or implicit correlations within a large collection of data. There are several sources to get data from, such as news websites, social media platforms (for example Facebook and Twitter), sensors, and other IoT (Internet of Things) devices. Social media platforms like Twitter prove to be important sources of data collection since the level of activity increases significantly during major events such as hurricanes, floods, and events of global importance.

For generating summaries, we first had to convert the WARC file which was given to us, into JSON format, which was more understandable. We then cleaned the text by removing boilerplate and redundant information. After that, we proceeded with removing stopwords and getting a collection of the most important words occurring in the documents. This ensured that the resulting summary would have important information from our corpus and would still be able to answer all the questions.

One of the challenges that we faced at this point was to decide how to correlate words in order to get the most relevant words out of a document. We tried several techniques such as TF-IDF in order to resolve this. Correlation of different words with each other is an important factor in generating a cohesive summary because while a word may not be in the list of most commonly occurring words in the corpus, it could still be relevant and give significant information about the event.

Due to the occurrence of Hurricane Irma around the same time as the occurrence of Hurricane Harvey, a large number of documents were not about Hurricane Irma. Accordingly, all such documents were eliminated, as they were deemed non-relevant. Classification of documents as relevant or non-relevant ensured that our deep learning summaries were not getting generated on data that was not crucial in building our final summary. Initially, we attempted to use Mahout classifiers but the results obtained were not satisfactory. Instead, we used a much simpler world filtering approach for classification which has eliminated a significant number of documents by classifying them as non-relevant.
We used the Pointer-Generator technique, which implements a Recurrent Neural Network (RNN) for building the deep learning abstractive summary. We combined data from multiple relevant documents into a single document, and thus generated multiple summaries, each corresponding to a set of documents. We wrote a Python script to perform post-processing on the generated summary to convert all the alphabets after a period and space to uppercase. This was important because for lemmatization, stopword removal and POS tagging, the whole dataset is converted to lowercase. The script also converts the first alphabet of all POS-tagged proper nouns to upper case.

ROUGE (Recall-Oriented Understudy for Gisting Evaluation) is used to evaluate the generated summary against the golden standard summary.

The results obtained were as follows:

- ROUGE para:
  1. ROUGE-1 : 0.16667
  2. ROUGE-2 : 0.0
  3. ROUGE-L: 0.11111
  4. ROUGE-SU4 : 0.025

- ROUGE sent:
  1. Max ROUGE-1 score among sentences : 0.84615
  2. Max ROUGE-2 score among sentences : 0.41667

- cov entity:
  Entity coverage:12.28 percent
2 Acknowledgement

We would like to express our acknowledgement and gratitude towards the following for guiding and assisting us throughout the course of the project.

- Dr. Edward A. Fox
- Liuqing Li
- Digital Library Research Laboratory
- NSF grant IIS-1619028: Collaborative Research: Global Event and Trend Archive Research (GETAR)
3 Introduction

3.1 Background

In the past ten years, there has been an exponential increase in the amount of information shared and posted online. The type of content available online ranges from textual articles to microblogs to images. In order to present an unbiased and objective view of an instance, it becomes crucial to filter information that pertains to an event but is not influenced by individualistic opinion.

One way to do this is to generate a summary of all the available information. Thus text summarization becomes a very important task to make sense of all the available data. Radev [1] defined a summary as “a text that is produced from one or more texts, that conveys important information in the original text(s), and that is no longer than half of the original text(s) and usually, significantly less than that”.

Humans can generate text summaries manually by reading the documents, understanding them, and then writing the key takeaways. Generating a text summary automatically, however, is much more difficult. Computers do not intrinsically possess the linguistic ability, the knowledge, or the context to generate a summary easily. We have to ensure that the summary is concise and that it reads fluently, while having all the relevant information and keeping the overall meaning.

While this is a difficult task, research has been conducted on this domain for decades. There are now two primary approaches to the task of generating a summary:

Extraction:
Extraction is the task of identifying the parts of the source texts that would be considered important, and directly extracting them with no modification. The summaries generated using this method are called extractive summaries.

Abstraction
Abstraction is the task of producing the information in a new way. This involves the use of advanced natural language techniques to understand the source texts and generate new shorter text containing the most important information. The summaries generated using this method are called abstractive summaries.
3.2 Framework

There is a generalized framework [21] for approaching any text mining or natural language processing task. The framework is visually demonstrated as a linear process, but in reality, it is rather iterative.

The high-level steps for the framework are as follows:

1. Data Collection or Assembly
2. Data Pre-processing
3. Data Exploration and Visualization
4. Model Building
5. Model Evaluation

The model that we will be building is text summarization for Hurricane Irma data corpus. Model evaluation involves the generated summary from the model being compared with a gold standard summary written for the same event.
4 Literature Review

As part of our literature review, we came across several Natural Language Processing tools like NLTK, Gensim, CoreNLP, SpaCy, TextBlob, and BeautifulSoup. Most of our work was done using NLTK and Gensim. The additional resources we used were GitHub repositories and research papers available on the internet.

4.1 NLTK

The Natural Language Toolkit, commonly referred to as NLTK, is a leading platform for building programs in the Python programming language to work with the data in human language. It consists of a suite of libraries and programs providing easy-to-use interfaces to over 50 corpora or lexical resources such as WordNet, as well as text processing libraries and wrappers for classification, tokenization, stemming, tagging, parsing, and semantic reasoning. It is used for symbolic and statistical natural language processing. It also has an active discussion forum [7].

4.2 Gensim

Gensim is a open-source vector space modeling and topic modeling toolkit. It uses NumPy, SciPy and optionally Cython, if performance is a factor. It is implemented in Python. Gensim includes implementation of tf-idf, word2vec, and document2vec algorithms, hierarchical Dirichlet processes (HDP), latent semantic analysis (LSA, LSI, SVD), and latent Dirichlet allocation (LDA) [2].

4.3 Classification Techniques

There are several classification algorithms that we came across which include Naive Bayes Classification, Decision Rules Classification, Logistic Regression, etc. We planned to use simple classification techniques as this was not the primary focus of our project.

- Naive Bayes Classification: It is a ‘probabilistic classifier’ based on Bayes theorem with strong independence assumptions among predictors, regardless of any possible correlation. It uses the method of maximum likelihood to determine the label [19].

- Decision Rules Classification: Rule-based classifiers make use of a set of IF-THEN rules for classification.
4.4 Deep Learning Techniques

There has been extensive research done in the field of deep learning to use neural networks for textual analysis. Using deep learning-based approaches for text summarization has gained significant attention. Initially, CNN-based approaches were tried and evaluated but they did not provide good results, since CNN cannot learn efficiently from time series data. The focus was then shifted to RNN based approaches; they provided efficient results. A majority of the initial research [10] revolved primarily around generating extractive summaries because of the ease of extractive summary generation as compared to abstractive text summaries [10][11][12]. However, the efficiency of sequence to sequence models [13], which use recurrent neural networks (RNNs) for textual analysis, has made abstractive summarization viable [13]. Rush [16] was the first to use a deep learning approach for abstractive text summarization [16]. Neural sequence to sequence models provided a viable approach for abstractive text summarization (meaning they are not restricted to simply selecting and rearranging passages from the original text). However, these models have two shortcomings: they are liable to reproduce factual details inaccurately, and they tend to repeat themselves. Abigail [18] implemented text summarization using Pointer-Generator networks which overcomes the shortcomings of a standard sequence to sequence model by copying words from the source text via pointing, which aids accurate reproduction of information, while retaining the ability to produce novel words through the generator. This model also makes use of coverage to keep track of what has been summarized, which discourages repetition [18].
5 Data Preprocessing

Preprocessing the data contained in the documents or web pages is basically performing the preparation tasks on the raw corpus in anticipation of a major text mining task. It is generally called data cleaning and it is the most important step in data mining. Sophisticated Natural Language Processing (NLP) techniques like lemmatization, stopword removal, removing boilerplate, and regex based string patterns removal can be used that help eliminate the noise present.

Figure 2: High-level view [22] of the iterative data preprocessing model.

More generally, some basic analysis and transformation are performed on the data corpus to be left with more meaningful and useful artifacts. This will be useful further, for the following important analytic task or core text mining or natural language processing work.

The dataset that we received was in the form of a WARC file that con-
tained all the information including the text, URLs, timestamps, etc. Our first task was to convert this WARC file into JSON format to make it more human readable and simplify our future steps.

To do this, we first stored the WARC file on the HDFS of the Hadoop cluster. Then, we executed the Scala script `script.scala` provided by the GTA to convert the data to JSON format. After generating the JSON file, we further formatted it, using another Python script `json_formatter.py` provided by the GTA, and generated a formatted JSON file. This formatted JSON file was then uploaded onto the Solr server for indexing.

We did this by doing the following steps:

1. Use ssh to connect to the Blacklight server
   
   ```
   ssh [USERNAME]@blacklight.cs.vt.edu
   ```

2. Copying the JSON file to our home directory on the Solr server using scp.

3. Copying the `json_formatter.py` file from `/home/fox` to our home folder on the Solr server using scp.

4. Running the `json_formatter.py` file to format the JSON file into a format suitable for Solr.
   
   ```
   python json_formatter.py [FILE]
   ```

5. Indexing by adding the file to the Solr server.
   
   ```
   ```

After the JSON file was generated, we used it for our future tasks instead of the originally provided WARC file as the JSON was easier to process and simpler to extract the relevant data from. Data processing involves a number of steps, some of which may or may not apply to our given dataset.
They basically fall under three broad categories or components:

1. Noise Removal
2. Normalization
3. Tokenization/Segmentation

![Text data preprocessing framework](image)

Figure 3: Text data preprocessing framework [21].

5.1 Noise Removal

The boundary between noise removal and data collection and assembly is very fuzzy. As our data corpus is built from collecting the data from the World Wide Web, it is usually in a raw web format, and there is a high chance that our text could be wrapped in HTML or XML tags. Depending on how the data was acquired, collected, and assembled, the metadata should be dealt with in the noise removal step. The data collected is usually in the JSON format and the required text should be extracted and the noise should be removed prior to tokenization or any other preprocessing steps. There are many pre-built existing software tools and pattern matching expressions that could be of help for removing the unwanted noise [3].
Noise removal involves the following:

1. Removing text file headers, footers

2. Removing HTML/Javascript code and “boilerplate” (headers, copyright notices, link lists, materials repeated across most pages of a site, etc.), markup, and metadata

3. Extracting valuable data from other formats, such as JSON, or from within databases

4. Regular expression matching to filter out unwanted text and task-specific noise

jusText is a tool for removing boilerplate content, such as navigation links, headers, and footers from HTML pages. It is designed to preserve mainly text containing full sentences and it is therefore well suited for creating linguistic resources such as Web corpora. [3]

![Figure 4: Code Snippet for jusText HTML cleaning.](image)

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Size in MB</th>
<th>Number of documents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Before noise removal</td>
<td>1532</td>
<td>14304</td>
</tr>
<tr>
<td>After noise removal</td>
<td>45</td>
<td>9942</td>
</tr>
</tbody>
</table>

Table 1: Results after noise removal using jusText

It is clearly evident from the Table 1 that the number of documents in the corpus has been reduced by almost 30 percent and size of the dataset has been reduced by 97 percent after using jusText program to remove boilerplate and other irrelevant information. It says that a significant number of documents in the corpus were just noise and completely unnecessary for generating the summaries.
5.2 Normalization

Normalization refers to converting all text to the same case (upper or lower case). We chose to convert all text into lower case. It basically is case-folding by reducing all letters to lower case, which puts all words on equal footing, leading to uniformity of the words.

Code for Normalization:

```python
text = text.lower()
```

5.3 Tokenization

Tokenization splits long strings of text into smaller tokens. A document can be tokenized into individual sentences, while sentences can further be tokenized into individual words.

```python
import nltk
from nltk.tokenize import sent_tokenize, word_tokenize
word_tokens = word_tokenize(text)
```

Figure 5: Code Snippet for tokenization.

This whole preprocessing step is not a linear process. Each of the steps could be applied in any order as per the requirement of our data corpus.
6 Data Exploration

6.1 Most Frequent Words

We initially attempted to extract the most frequent words by simply tokenizing each word in the data and generating the frequency distribution for each word. This result of this was that we ended up with many articles, prepositions, and punctuation symbols as our most frequent words. We then optimized this approach by first filtering out the punctuation and special characters from our data by using regular expressions.

![Image of most frequent words after tokenization.](image)

Figure 6: Most frequent words after tokenization.

Next we removed the frequent stopwords and many of the irrelevant words from the data by using NLTK’s library of stopwords for the English language. The same words, with different cases, were combined by converting all word tokens to lowercase, to get the actual frequency of those words. Consequently, we were able to generate a much better list of most frequent words that were more indicative of our data collection.

```python
nltk.download('stopwords', 'english')
stopWords = set(stopwords.words('english'))
```
But we could still see a few words in the frequency distribution that were commonly used in the English language and could be removed from the data. So, we decided to work on our custom stopwords list in addition to that from NLTK and were able to obtain a much more indicative list of most frequent words for our data collection.

```
custom_stopwords = ['would', 'could', 'said', 'u']
```

Figure 7: NLTK stopwords.

Figure 8: Most frequent words after stop word removal.
One drawback of following this approach was that when we converted all words to lowercase, all proper nouns were converted to lower case as well, which resulted in difficulties in named entity extraction.

### 6.1.1 Lemmatization

After generating the list of frequent words, we realized that were many words in the distribution that were different forms of the same root word. So, we decided to convert each word to its root form before generating the frequency distribution. We used NLTK’s lemmatizer function to change all words to their root form and then proceeded to get the updated frequency distribution.

Code for Lemmatization:

```python
wnl = nltk.WordNetLemmatizer()
w = wnl.lemmatize(w)
```

An issue we faced after performing lemmatization on the dataset was that when we later attempted to perform parts of speech tagging, the parts of speech that were returned for the stemmed words were incorrect, which could have been due to the incorrect grammar of the sentences after lemmatization.

### 6.1.2 POS Tagging

After generating a list of most frequent words, we attempted to extract the parts of speech of the words and generate a list of frequent words constrained by their part of speech. We first used the punkt tokenizer to get sentence tokens that we passed to NLTK’s POS tagger. Upon getting the results, we realized that the results were highly inaccurate. After digging deeper we concluded that the sentence structure of the entire dataset had been corrupted. This was because lemmatization had changed all words to their root forms, we had converted all words to lowercase, and we also had removed all stopwords, punctuation, and articles. Thus, the sentence tokens passed to the NLTK POS tagger were incorrect, which led to incorrect results.

We fixed this by passing the unfiltered data to the POS tagger; we immediately observed words tagged with their correct part of speech.
from nltk.corpus import wordnet as wn
morphytag={‘NN’:wn.NOUN,‘JJ’:wn.ADJ,’VB’:wn.VERB,’RB’:wn.ADV}
wnl = nltk.WordNetLemmatizer()
pos_tuple = nltk.pos_tag([‘word’])
pos_tag = (pos_tuple[1][1]:[2])
tag = morphy_tag(pos_tag) if pos_tag in morphy_tag else None
w = wn1.lemmatize(word, tag) if tag else wn.lemmatize(w)

Figure 9: Code Snippet for Lemmatization using POS tagging.

[(‘hurricane’, 47767), (‘irma’, 38625), (‘storm’, 27288), (‘florida’, 27021),
(‘people’, 18916), (‘home’, 15908), (‘get’, 12763), (‘make’, 12490), (‘island’,
12382), (‘damage’, 12142), (‘one’, 11951), (‘water’, 11891), (‘go’, 11859),
(‘state’, 11713), (‘power’, 11573), (‘wind’, 11393), (‘help’, 11288), (‘take’,
10024), (‘day’, 9642), (‘county’, 9632), (‘also’, 9578), (‘2017’, 9393), (‘time’,
9219), (‘area’, 9184), (‘need’, 9043), (‘work’, 8472), (‘flood’, 8466), (‘year’,
8344), (‘us’, 8291), (‘come’, 7658), (‘st’, 7784), (‘many’, 7565), (‘like’,
7328), (‘resident’, 7279), (‘see’, 7030), (‘say’, 6966), (‘vietnam’, 6830),
(‘center’, 6815), (‘still’, 6738), (‘september’, 6710), (‘hit’, 6615), (‘wall’,
6596), (‘sept’, 6497), (‘include’, 6468), (‘two’, 6464), (‘emergency’, 6462),
(‘week’, 6438), (‘new’, 6373), (‘caribbean’, 6343), (‘monday’, 6291), (‘know’,
6280), (‘back’, 6260), (‘city’, 6216), (‘shelter’, 6190), (‘disaster’, 6180),
5923), (‘million’, 5897), (‘keys’, 5844), (‘community’, 5692), (‘even’, 5680),
(family’, 5641), (‘south’, 5596), (‘house’, 5549), (‘news’, 5452), (‘sunday’,
5362), (‘national’, 5362), (‘high’, 5332), (‘move’, 5285), (‘service’, 5257),
(‘way’, 5243), (‘u’, 5172), (‘call’, 5152), (‘10’, 5071), (‘puerto’, 5024),
(‘west’, 4938), (‘category’, 4912), (‘tree’, 4890), (‘may’, 4808), (‘provide’,
4773), (‘local’, 4709), (‘last’, 4609), (‘much’, 4583), (‘part’, 4576), (‘key’,
4567), (‘expect’, 4558), (‘show’, 4479), (‘north’, 4429), (‘change’, 4411),
(‘official’, 4355), (‘around’, 4308), (‘across’, 4266), (‘place’, 4242),
(without’, 4240), (‘hour’, 4224), (‘life’, 4199), (‘rico’, 4188)]

Figure 10: Most frequent words after lemmatization using POS tags.
6.2 Bigrams

There are many pairs of words which occurred together like ‘hurricane irma’, ‘puerto Rico’, ‘virgin islands’, etc.
6.3 Named Entity Extraction

Next, we went on with extracting named entities from the data. We again used the unfiltered data for this because the filtered data was all lowercase and had word stems instead of the actual words. We had the choice to go with either the Stanford NER or the NLTK NER, and we decided to go with the latter as it seemed to give better results. The Stanford NER mistagged many nouns and verbs and did not recognize ‘Irma’ as a noun at all.

We passed the words along with their part of speech to the NLTK NER and observed satisfactory results. There were many words that were incor-
rectly tagged like ‘couldn’t’ was tagged as a noun. But such instances were very few and we decided to go ahead with this approach.

Figure 15: Named entities.

6.4 LDA Topic Modeling

A topic model is a statistical model that aids in the discovery of abstract “topics” occurring in a collection. If a document is about a topic, we would expect that certain words would appear in the document frequently. Latent Dirichlet Allocation (LDA) is a way of automatically studies documents, that can be sentences or passages or whole papers. It aims to find the topics across the collection of documents, such that each document is represented as a mixture of one or more of those topics. We used gensim’s corpus and LDA model to generate the topics for our dataset. We initially ran the LDA model on the unfiltered data and generated 10 topics with different counts of words in each and concluded that having too many words in the topic leads to inclusion of non-relevant words. So, we decided to set the count of words as 4 before proceeding ahead.

```python
NUM_TOPICS = 10
ldamodel = gensim.models.ldamodel.LdaModel(corpus,
num_topics = NUM_TOPICS, id2word=directory, passes=15)
ldamodel.save('model5.gensim')
topics = ldamodel.print_topics(num_words=4)
```

Figure 16: Code Snippet for LDA Topic Modeling.
Figure 17: Topics generated by LDA topic modeling.
7 Classification

7.1 Mahout Classifier

Since there was a very large amount of data available to us, it was not surprising that most of the documents were not relevant at all to Hurricane Irma. We realized that it is important that we classify the documents in the data corpus into relevant and non-relevant to Hurricane Irma and use only those documents that are relevant to ensure that the abstractive summary does not contain any insignificant information. The duration of Hurricane Irma was approximately from August 30, 2017 to September 13, 2017 while that of Hurricane Harvey was from August 17, 2017 to September 2, 2017. Since these two hurricanes occurred in overlapping time frames, a lot of our data only has information about Hurricane Harvey and just mentions Hurricane Irma briefly. It is important that we remove such documents which are not relevant to Hurricane Irma. Also, there was a significant number of duplicate documents in the corpus.

```python
lines_seen = set()
outfile = open('cleaned_dup_removed.txt', 'w')
for line in open('cleaned_text.txt', 'r'):
    if line not in lines_seen:
        outfile.write(line)
        lines_seen.add(line)
outfile.close()
```

Figure 18: Code Snippet for duplicate removal.

A lot of documents in the corpus also were mostly about personal experiences of people during the hurricane. It was a trade-off where we could either eliminate such documents and lose the important information in those documents or keep the documents with some non-relevant information associated with them. We decided not to eliminate them because they still contained some relevant information about Hurricane Irma.

Mahout allows the users to choose from three classification algorithms.

1. Complement Naive Bayes
2. Naive Bayes
3. Stochastic Gradient Descent
We decided to use Mahout’s Complement Naïve Baye’s classifier to classify our documents. We manually labelled 300 documents as relevant and irrelevant and since our data set comprised mostly of irrelevant documents, our training set also had a slightly larger number of irrelevant documents. First, we converted the labelled dataset into a sequence file using Mahout’s ‘seqdirectory’ command. Then we used ‘seq2sparse’ to generate tf-idf vectors from the sequence files.

```
mahout seqdirectory -i ${WORK_DIR} -o ${WORK_DIR}/seq -ow

mahout seq2sparse -i ${WORK_DIR}/seq -o ${WORK_DIR}/vectors
-lnorm -nv -wt tfidf
```

Next we split the vectors into a training and a testing set, and trained the model on the training set. Upon running the model on the test set, we observed approximately 71% accuracy. But when we ran the model on the entire big data, we observed that only 2 documents were classified as irrelevant and all the rest were marked relevant. Seeing such results we dropped the idea using the Complement Naïve Baye’s classifier and started looking for alternate approaches to classify our dataset.

```
mahout split -i ${WORK_DIR}/vectors/tfidf-vectors
--trainingOutput ${WORK_DIR}/train-vectors
--testOutput ${WORK_DIR}/test-vectors
--randomSelectionPct 40
--overwrite --sequenceFiles -xm sequential

mahout trainnb -i ${WORK_DIR}/train-vectors -el -o ${WORK_DIR}/model
-li ${WORK_DIR}/labelindex -ow -c

mahout testnb -i ${WORK_DIR}/test-vectors -m ${WORK_DIR}/model
-li ${WORK_DIR}/train/labelindex -ow -o ${WORK_DIR}/test/
  test_results -c
```

Figure 19: Code Snippet for Classification using Mahout.
7.2 Decision Rules Classifier

Since the Mahout Complement Naïve Baye’s was not able to successfully label irrelevant documents, we decided to use a Decision Rules type of classification approach. Applying Occam’s razor, we chose the model that is the simplest to interpret, to explain, to deploy, and to maintain. The decision rule used was if a document does not contain the words ‘Irma’ or ‘irma’, then the document is marked non-relevant, otherwise it is marked relevant. Since our data corpus was on Hurricane Irma, it was safe to assume that any document that contains information relevant to Hurricane Irma should contain at least one of the words ‘Irma’ or ‘irma’.

Upon completion of the filtering process, the dataset was reduced to approximately 7000 documents, i.e., about 50% of our original dataset.
outfile = open('cleaned_noise_remove.txt', "w")
with open("cleaned_dup_removed.txt") as fin:
    for line in fin:
        if 'irma' in line or 'Irma' in line:
            outfile.write(line)
        outfile.close()
8 Clustering

We attempted to perform clustering on the data set to see if we could cluster similar documents together and then try to generate summaries cluster-wise. We also hoped that this would cluster the irrelevant documents separately from the relevant documents or mark them as outliers so that we could end up with a filtered dataset for summarization. We used Mahout’s k-means clustering algorithm, setting it to begin clustering with 10 random initial documents over 20 iterations. The clustering was completed successfully, but when we attempted to use Mahout’s clusterdump command to get the results in a text file, the Hadoop cluster threw memory errors every time. Since, we were heading towards the end of the semester and did not have much time left, we shifted our focus towards generating abstractive summaries.

From the partial results that were obtained, we could see that the clusters that were formed contained similar information and also that there were clusters that contained no relevant information at all.

Unfortunately due to the time constraint, we were not able to take this further and went ahead with other means for the summarization process.

```
mahout kmeans -i ${WORK_DIR}/vectors/tfidf-vectors/ -c
${WORK_DIR}/clusters -o ${WORK_DIR}/kmeans -dm
org.apache.mahout.common.distance.EuclideanDistanceMeasure
-x 10 -k 20 -ow --clustering

mahout clusterdump -i ${WORK_DIR}/kmeans/clusters-*.final -o
/home/cse4984ca5964f18_team9/kmeans/clusterdumps/clusterdumps.txt
-d ${WORK_DIR}/vectors/dictionary.file=0 -dt sequencefile=-b 100
-n 20 --evaluate -dm
org.apache.mahout.common.distance.EuclideanDistanceMeasure
-sp 0 --pointsDir ${WORK_DIR}/kmeans/clustedPoints
```

Figure 22: Code Snippet for Clustering using Mahout.
Figure 23: Relevant clustered documents.

Figure 24: Irrelevant clustered documents.
9 Abstractive Summaries

Summarization is the task of condensing a piece of text to a shorter version that contains the main information from the original. Summarization can be broadly classified into two types: abstractive and extractive summarization. Extractive methods assemble the summaries by taking whole sentences from the source text. On the other hand, abstractive methods may generate novel words and phrases that are not present in the source text. The extractive approach is naturally a lot easier than the abstractive approach since the baseline sentence structure and grammar is maintained in copying chunks of data from the source text. On the other hand, sophisticated techniques that are important in high quality summarization such as paraphrasing and generalization are possible only in an abstractive framework [18].

![Sequence to sequence model.][18]

Abstractive methods can attend to relevant words in the source text and then generate novel related words. For instance in Figure 16, to produce the word ‘beat’ in the abstractive summary from the sentence ‘Germany emerge victorious in 2-0 win against Argentina on Saturday’, the model may attend to the words ‘victorious’ and ‘win’ in the source text. [18]

In order to generate abstractive summaries from provided data, we decided to make use of a deep learning model called Pointer-Generator Network. Neural sequence to sequence models [18] have provided a viable new approach for abstractive text summarization. However, these models have two shortcomings: they are liable to reproduce factual details inaccurately, and they tend to repeat themselves. [18]
The model in Figure 17 calculates a generation probability which weighs the probability of generating words from the vocabulary, versus copying from the source text. The distribution is weighted and assumed to obtain the final distribution from which the prediction is made.

For this reason, we chose to use a Pointer-Generator Network that augments the standard sequence to sequence attentional model in two orthogonal ways. Pointer-generator network is a hybrid between point-to-point and a pointer network as it allows both copying words via pointing, and generating words from a fixed vocabulary. The hybrid model can copy words from the source text via pointing, which aids accurate reproduction of data while retaining the ability to produce novel words through the generator. The data that has been summarized is constantly kept track of in order to avoid repetition. [18]

9.1 About the Model

As shown in Figure 17, generation probability is used in the Pointer-Generator model in order to choose between generating a word from the vocabulary by sampling the vocab file generated from preprocessing the data, or copying a word from the input sequence by sampling from the attention distribution. The attention distribution can be viewed as a probability distribution over the source words, that tells the decoder where to look to produce the next word. The ability to produce out of vocabulary words is one of the primary
advantages of Pointer-Generator models. The Pointer-Generator model also solves a commonly occurring problem in sequence to sequence models: repetition. Repetition is especially pronounced when generating multi-sentence text. The Pointer-Generator model makes use of a coverage model that maintains a coverage vector which is the sum of attention distributions over all the previous steps. Intuitively, this vector represents the degree of coverage that certain words have received form the attention mechanism thus far. The coverage vector acts as an extra input to the attention mechanism to ensure that the attention mechanism’s current decision is informed by a reminder of its previous decisions to make it easier for the attention mechanism to avoid repeatedly attending to the same locations, thus avoiding repetitive text in the generated summaries.

9.2 Preprocessing the Data

One of the biggest challenges that we faced in order to utilize the Pointer-Generator network to get abstractive summaries was to preprocess our data in the same format that is suitable for the network. Firstly, we separated each document in the cleaned JSON file into a separate .story file each having a unique name. So the number of .story files we have is equal to the number of documents we have in the cleaned JSON file. Our initial idea was to generate a hash based on the URL of each document and use the generated hash as the name of the corresponding .story file of the document. But since the URL of the document is not useful for generating the summary, we had removed the URL from the initial JSON itself to get a cleaned JSON. So instead of making it complicated by hashing the URL and using it as the title of the .story file corresponding to the document, we named the .story file just based on the index of its corresponding document in the JSON file. In short, the first document in the JSON file will be named as 1.story, second document as 2.story, third document as 3.story, etc. In this way we ensured that each .story file will be named uniquely.
Once we have the stories of all the documents, we tokenize the stories using the Stanford tokenizer. Stanford CoreNLP is a natural language software toolkit which provides a set of human language technology tools, including for grammatical analysis. It simplifies the summarization by converting the words to their base forms; performing part of speech tagging; normalizing dates, times, and numeric quantities; grouping similar noun phrases; marking the quotes within sentences; etc. It does so for all the stories, and generates a corresponding tokenized story for each story generated from the raw cleaned JSON file in the previous step. After generating the tokenized stories, we process the data into binary (.bin) and vocab files using TensorFlow. The binary files and vocab files are used by the Pointer-Generator network to generate summaries.

Figure 27: Code snippet to get .story files from the input JSON file.

Figure 28: Output snippet when generating binary files and vocab files from the .story files
9.3 Training the Model

For the intent of this project, we worked on a pre-trained Pointer-Generator model that was trained on a CNN/Dailymail dataset. We preserved the vocab files that came with the dataset and did not insert the vocab file from our dataset since the vocab file is crucial to the dataset that the model has been trained on. For getting our final summaries, however, we trained the model with our own dataset, and plugged in the vocab file generated from preprocessing the Hurricane Irma dataset, in order to get better results. When creating the binary files in the data preprocessing stage, we have divided the overall data we have into train data, test data, and validation (eval) data with their respective percentages being 50, 40 and 10, and create separate binary files. An LSTM neural network model is created using TensorFlow. The final Pointer-Generator model runs in three modes, namely: train mode, eval mode, and decode mode. When the train mode is running, the model trains on the train.bin files and creates all the required checkpoints, events, model metadata, and vocab metadata. The evaluation mode is used to test the trained model based on the loss on the validation (eval) dataset. By running the evaluation mode, we can make sure that the training model does not overfit or underfit. We can select a model which provides the least loss on the validation dataset. The decode mode is where we generate the final summary. We use the test dataset to generate the summary. We must use the same hyper-parameters as used while training the model and use the vocab file, checkpoints, and all the other data generated while training the data in the decode mode to generate the final summary.

9.4 Post-Processing the Data

Upon careful analysis of our summary, we noticed a lot of words in the generated summary were taken from the vocab file generated from preprocessing the data, that did not make sense in the sentence structure. We preprocessed the data by adding these words as custom stopwords and filtering the generated summary to carry out the cleaning process. An example of this is “The national hurricane center -lrb- nhc -rrb- released a tropical cyclone report that outlined the history, meteorological statistics, casualty and damage statistics.” Here, ”-lrb-“ and ”-rrb-“ are unwanted words that were picked up by the vocab file that envelope a meaningful and important ‘nhc’ in the sentence. Apart from that, since our data cleaning process involved converting all the words to lowercase, we wrote another Python script to convert the first letter after every period and space to uppercase. Another challenge that we faced with our generated summaries was proper nouns
being lowercase. We decided to automate this process by writing a script to capitalize the first letter of all POS tagged proper nouns. This did not provide adequate results for terms that were not classified as a bigram. For instance, dominican republic was changed to Dominican republic but we also want ‘r’ in republic to be capitalized in the final summary. As part of future work, our intent is to include all states and countries in bigrams and trigrams before running our capitalization script over the summary. Since the generated deep learning summary was longer than two pages, we did manual post-processing like removing a few sentences which provided the same information but in a different sentence formation, removing information regarding personal experiences of people, etc. to cut down the summary to two pages.

9.5 Hurricane Irma Gold Standard By Team 8

The following was provided to us.

- Team 8: Xiaoyu Chen, Maanav Mehrotra, Haitao Wang, Di Sun, Nakman Chhikara

Hurricane Irma was a Category 5 Atlantic hurricane, the most powerful in history, with winds of 185 mph for 37 hours, longer than any storm recorded. Tropical force winds extended 185 miles from the center. Irma held 7 trillion watts of energy. Storm surges brought waves 20 feet higher than normal. It hit the Caribbean and the United States, causing damage of over $64 billion, making it the fifth most costly storm. Hurricane Irma took 129 lives and left hundreds of people injured.

The storm began as a weak wave of low pressure off the west African coast on 27 August. It formed just west of the Cape Verde Islands on 30 August. Fueled by above average (in the mid-80s F) sea surface temperatures, Irma increased in intensity to a Category 3 by late August 31, with 115 mph sustained winds, and by September 5 to Category 5, continuing that for 3 days. At near peak strength as it approached the Leeward Islands, it made landfall along the northern coast of Barbuda, damaging 90% of the buildings, but with less severe damage in Antigua, nevertheless causing over $150 million losses on those islands, and devastating wildlife. Irma’s pressure bottomed out at 914 Mb. Maintaining its intensity, it made landfalls September 6 on Saint Martin (where over 90 percent of the structures were damaged, coupled with ripping out trees, ruin of marinas, and over $4 billion in losses), Sint Maarten (with an estimated $1.5 billion losses), and Virgin Gorda in the British Virgin Islands (where it caused massive defoliation that was apparent even from space), also affecting Saint Barthelemy with flooding. It passed north of Puerto Rico on September 6 and Hispaniola on September 7. On September 8 it passed south of the Turks and Caicos Islands, and
weakened to Category 4, but devastated parts of the southern Bahamas with 155 mph winds, and intensified to Category 5 with 160 mph winds before landfall in Cuba, after which it weakened to Category 2 on September 9. It strengthened to Category 4 on September 10, making landfall in Cudjoe Key, Florida, with 130 mph winds. It weakened to Category 3, making its final landfall in Marco Island, Florida, with 115 mph winds. And then it down to a Category 2 storm with sustained winds of 110 mph by late September 10.

Irma caused a large amount of rainfall. The maximum recorded precipitation was 23.90 inches in Topes De Collantes, and 21.66 inches in St. Lucie County in the United States. The average rainfall in the affected areas was around 10-15 inches.

Power outages were widespread, in Anguilla, the Lesser Antilles, U.S. Virgin Islands, Puerto Rico (for over a million residents), the Bahamas, and USA (with more than 9 million outages). It left Puerto Rico without power. Saint Martin, It disconnected the public transportation and bridges, and damaged properties of more than 70 million people’s. Barbuda was left with no water or communications after the storm. Coming to Florida, there were widespread power outages and flooding. It brought 21 storms in Florida and caused much more damage. Before hurricane Irma hit, officials had a minimum of 3-4 days for warnings and evacuations. Most of the Caribbeans (i.e., Puerto Rico, Antigua and Barbuda, Guadeloupe, Turks and Caicos, Haiti, Bahamas, Dominican Republic, and Cuba) and Southern states of US (i.e., Florida, Georgia, and parts of South Carolina) were issued with warnings or evacuated. In the Caribbean islands, Puerto Rico declared a state of emergency on September 4. In the Dominican Republic, 112,000 people were evacuated from vulnerable areas. Another 7400 tourists were moved to Santo Domingo. In the Bahamas, 1,609 people were evacuated by air from the southern islands, including 365 from Bimini. The government began preparation a week ago by securing national sports facilities to use as shelters. About 1200 were housed at the Atlantis Paradise Island. In the United States, Florida was the first state to declare a state of emergency on September 4. The Florida Keys, Jupiter Inlet to Bonita Beach were the first places to issue watches and warnings on September 7. There were around 700 emergency shelters set up for 191,764 people. Georgia started issuing warnings from September 9. Governor Nathan Deal declared a state of emergency on September 8. Around 540,000 people living near the coastline and east of I-95 were mandatorily evacuated. North Carolina and South Carolina also declared a state of emergency on September 6 to mobilize resources. After the hurricane, teams of experts from the Federal Emergency Management Agency and other federal agencies formed the Interagency Recovery Coordination group. The US Department of Children and Families provided $1
billion in food aid to Florida. Electric power for 95% of customers was re-
stored in Florida by September 18. The International Rescue Committee and
the Red Cross helped out in helping rebuild and distributing relief items.

9.6 Generated Summary

The final abstractive summary is shown below. This is generated by gaining
more insights on the data by exploring it as discussed in section 6, performing
data preprocessing to have a clean and uniform data as discussed in section 5,
performing classification for filtering out relevant and irrelevant documents
as discussed in section 7 and using a pointer generator network programmed
in Python using Tensorflow as discussed in section 9.2 and 9.3 and perform-
ing some post processing as discussed in section 9.4.

Tropical cyclone Irma developed on 28 August 2017 from a tropical wave
in the Atlantic that originated in Africa. As it moved west, it intensified
with the help of exceptionally warm ocean waters to become a category 5
hurricane on the saffir-simpson scale, with peak wind speeds of 300 km/h.
The storm’s center was about 650 miles west of the Cabo Verde islands off
the west coast of Africa. It was heading west-northwest at 10 mph, and
no coastal watches or warnings were in effect. Hurricane Irma pummeled
St Thomas and St John in the US Virgin Islands before it continued on its
devastating path to Puerto Rico.

The National Hurricane Center (NHC) released a tropical cyclone report
that outlined the history, meteorological statistics, casualty and damage statis-
tics. The report outlined that Hurricane Irma’s strong winds, heavy rains,
and high surf directly caused 44 fatalities in the Caribbean and the south-
eastern United States. Hurricane Irma, the most powerful Atlantic hurricane
in recorded history, hit the islands of the northeast Caribbean on Sept. 6,
2017, before roaring along a path to Puerto Rico, the Dominican Republic,
Cuba and Florida.

Hurricane Irma made landfall on September 10 as a category 4 hurricane
at 9:10 a.m. on Cudjoe Key, with wind gusts reaching 130 mph. States of
emergency were also issued in Alabama, Georgia, North Carolina and South
Carolina. Hurricane Irma made another landfall in Naples, Florida. Irma,
one of the strongest hurricanes on record in the Atlantic basin, made landfall
a total of seven times. The storm gradually lost strength, weakening to a
category 1 hurricane by the morning of September 11, 2017. At 5 a.m. ET,
Irma was carrying maximum sustained winds of nearly 75 mph. After days
of frantic preparations, residents in Tampa were bracing for Irma’s arrival,
weaker than expected but still packing wind gusts of about 100 miles per hour. The storm continued to lose strength as it pushed inland, but its reach extended from South Florida to Jacksonville. The powerful storm left around 5.8 million homes and businesses in Florida without power. Florida’s tourism industry is assessing the damage as the state heads into the busy holiday and winter travel bookings season. Officials were estimating the damage could top $100 billion. The death toll from Irma is estimated to be 129 people. Eight people have been killed and 23 injured in French island territories. Irma left about 6.7 million people without power. Irma left 5.5 million of the more than 10 million Florida power customers waiting for the electricity to be turned back on, including more than 834,000 in Tampa bay. More than 1.2 million customers in Georgia were without power after Irma swirled into the state. At least 7,000 flights canceled as the devastating category 4 hurricane neared closer to Florida’s southern tip. Royal Caribbean announced it is forced to cancel the September 11 sailing of Enchantment of the Seas due to the continued closure of port Miami. Enchantment of the Seas was scheduled to be a 4-night cruise that departed on September 11. Downtown Charleston and most coastal areas had a lot of flooding. Heavy rains caused flash flooding in parts of the Artibonite valley and its mountains, sweeping away gardens, fruit trees, damaging and destroying homes. Quarter of all homes in Florida keys are believed to be destroyed.

Around 6.3 million Floridians are under evacuation orders. Over 20 counties are being told to evacuate, in what could be the largest evacuation in American history. Mandatory evacuations were ordered for Miami-Dade and Palm Beach. At least 35 hospitals in Florida, Georgia, and South Carolina have either closed entirely or ordered partial evacuations. Evacuations were ordered in Jacksonville, Hilton Head Island and six other barrier islands. 540,000 people of Georgia were ordered to leave. Mandatory evacuations were issued to Chatham County and areas east of I-95. MIT has launched riskmap, a crowdsourced platform meant to track and map flooding by relying on people’s social media reports. Google’s emergency response team will mark the closed roads on Google maps in real time to assist the evacuation. These road closures will also appear on Irma crisis map, embedded as part of sos alert on search. More than 600 shelters are open in Florida with a population of more than 162,000. By Sunday afternoon, some 15 to 20 families were sheltered at leader’s preparatory school, a private Muslim school that had become one of three area shelters operated by the Islamic society of central Florida. Schools served as most of the 42 shelters that Miami-Dade’s county government opened ahead of Irma.

A $ 15.3 billion hurricane relief bill sailed through the senate and the house and was signed by president trump on Friday on Sept. 8. President Donald
trump signed the disaster-relief package in February. Florida’s department of transportation will receive $25 million in federal funding for relief efforts in the wake of the damage caused by hurricane Irma. Walmart donates $10 million. The Walt Disney Company donates $2.5 million. Wells Fargo is donating a total of $1.1 million. South of the interchange with state road 874, will remain toll-free to assist Monroe County with recovery. Wasp, the first navy platform to arrive in the vicinity of the U.S. Virgin Islands, is providing medium and heavy lift helicopters to transport people and supplies. Nearly 100 employees from Mississippi will assist Georgia power with their restoration efforts following hurricane Irma. Key West officials anticipate their island will be open by fantasy fest, the annual costume festival, on Oct. 20. More than 8,000 personnel continue to work to restore power to the 158,000 Georgia people who are still without power. Due mainly to the widespread loss of power, cell phone service was also impacted after battery backup power for cell phone towers ran out and backup generators ran out of fuel. More than 60000 personnel are activated from more than 250 electric companies, public power utilities, and electric cooperatives, who are dedicated to power restoration. About 70 percent of power is restored across affected areas by Sept. 15.

FEMA transferred approximately 7.2 million meals, 5.5 million liters of water, and 41 generators at the states’ request. Florida department of transportation has spent $15 million on debris removal from state highways. Governor Rick Scott announced Sept. 13 that the federal highway administration is providing emergency funds to help the state pay for immediate road and bridge repairs. Miami international airport reopened at 4 a.m. Tuesday. Taxi and transportation network company service resumed Wednesday. The Florida department of transportation also repaired two stretches of the highway that had washed away during hurricane Irma two days earlier at mile marker 37 and mile marker 75.

9.7 ROUGE Evaluation

ROUGE is a summary evaluation toolkit. It uses the ROUGE (Recall-Oriented Understudy for Gisting Evaluation) system of metrics to compare an automatically generated summary against reference summaries [20]. We used a Python wrapper of the ROUGE toolkit [9] to evaluate our generated summary.

The results were as follows:

- ROUGE\_para:
  
  ROUGE-1: 0.16667
As can be seen from the results above, the generated abstractive summary when compared with the Gold Standard provided by Team 08, did not perform well when a paragraph wise comparison was performed using ROUGE_para. But on sentence wise comparison using ROUGE_sent, the scores are high, which could have been due to different paragraph structures in the Gold Standard and the generated summary. Another reason for this could be writing style used in the Gold Standard which would depend on the writer of the Gold Standard.
10 Gold Standards

10.1 Introduction

Our team was tasked with creating a Gold Standard text summary for Team 12. The objective of this task was to work as a team and collectively summarize a large dataset into a coherent overview of all the relevant and important facts detailed in the entire collection. Team 12’s dataset had to do with Hurricane Florence. Since our team had also been working on a hurricane dataset, we had a fair idea of what would be needed to create the gold standard, and where those answers would be found.

We made use of the resources provided to us in Canvas, particularly the Hurricane Metadata page, which contained a list of questions that a good summary should answer about a hurricane. These questions were partitioned into different categories because of which we could write summaries over multiple smaller subsets of questions, and then later merge them together.

10.2 Exploring the Data

To create the summary, we needed to find the answers to the questions we needed. The ways we did this are discussed in the next two subsections.

10.2.1 Solr

Solr, from the Apache Lucene project, is an open source enterprise search platform [4]. It is scalable, fault tolerant, and reliable.

To search and return the results of a query, Solr does these four operations:

1. Indexing: The documents are converted to a format that’s easily readable for the machine.

2. Querying: Understanding the user’s query to understand what the user wants to retrieve

3. Mapping: The user query is mapped to the indexed database to generate a result.

4. Ranking: The outputs are ranked based on estimated relevance.

At the start of the course, all of the teams had created Solr indexes for their respective datasets as mentioned in Section 5 on the Solr server provided to us for this course at http://blacklight.cs.vt.edu:8983/solr/.
Our first step in exploring the data was to visit the Solr server. Here, we found the core where Team 12 had uploaded their dataset (big_team12).

![Solr Dashboard](image)

**Figure 29: Solr Database**

We identified that this dataset contained 10,948 documents, which would take a long time to sift through individually. We instead made use of Solr’s Query page, which let us look only for the documents relevant to the information we needed. This page contains a lot of filters that can be used to retrieve only the documents that fit the criteria after applying the set filters.

For instance, when we needed to look at information about the state of public transportation, adding the text “Sentences_t:"public transportation"” to the field “fq” (filter query) [4], with double quotes added around the term “public transportation” to ensure they appeared together, brought down the number of documents to look through to just 7.
From here we were able to look through the corresponding webpage with that URL to read the text and get the information we needed.

10.2.2 The Internet

Occasionally we would run into circumstances where the information we needed wasn’t easily retrievable through Solr. In this case, a solution we employed was to simply look up the information on the Internet. Hurricane Florence was a recent hurricane, so all of the online news media were still posting articles about it with updated and accurate information. We also realized that the Wikipedia page for the hurricane was in itself a summary of the event, so it too became a resource to find some of the information.

10.3 Gold Standard for Team 12 (Hurricane Florence)

We have formulated all the answers which we thought are necessary for summarizing an event after carefully exploring and examining the dataset and using the internet as discussed above. We have included all these in the gold standard. The sentences picked up from the dataset were not changed and included as it is in the gold standard. But for the information extracted from the internet, we have used our own sentence formation to include them in the gold standard.

The National Hurricane Center identified a potential tropical storm in the eastern Atlantic Ocean with a wind speed of around 30 mph on August 30,
2018. It originated near Cape Verde, off the coast of West Africa. This became a tropical storm named Florence on September 1. It developed into a Category 2 Hurricane on September 4. On September 5, it was promoted to Category 3 hurricane with maximum wind speed around 130 mph. By September 7, it had degraded to a tropical storm. However, it was fueled by 29 degree C sea surface temperatures, so on September 10, it strengthened into a Category 4 hurricane with sustained wind speeds of 140 mph and a central pressure of 939 mbar. On September 12, Florence’s peak wind speeds gradually fell to 110 mph, downgrading the hurricane to Category 2. Hurricane Florence finally made landfall on September 14 as a Category 1 hurricane with sustained winds of 80 mph near Wrightsville Beach, North Carolina at 7:15 am local time. The storm was moving at just 6 mph when it landed and had maximum sustained winds near 90 mph. By the end of the day, Florence was downgraded to a tropical storm and was moving slowly west into South Carolina at just 3 mph. The hurricane cut a path through the Eastern United States, particularly North Carolina and South Carolina. Florence weakened as it moved inland, degenerating into a post-tropical cyclone over West Virginia with maximum sustained winds of 25-30 mph on September 17, and dissipating on September 19 over the North Atlantic.

Hurricane Florence was the wettest tropical storm cyclone on record in the Carolinas, causing severe damage due to storm surge and freshwater flooding. Florence caused a maximum of 36 inches of rainfall in Elizabethtown, North Carolina, 34 inches in Swansboro, and 24 inches of rainfall in both Wilmington, NC and Loris, South Carolina. An average of over 20 inches of rainfall was recorded covering 14,000 square miles from Fayetteville, North Carolina, to Florence, South Carolina. Major rivers like Neuse, Eno, Cape Fear, and Lumber spilled over their banks, with inland flooding in Fayetteville, Smithfield, Lumberton, Durham, and Chapel Hill.

The death toll from Hurricane Florence was at least 55. Of them, 30 were a direct consequence of the hurricane. Florence caused immense damage to public and private property. CoreLogic estimated over 600,000 residential homes were damaged by flooding and/or winds. The total cost of the damage was estimated to be between 38 billion and 50 billion dollars.

From 7-12 September, the mayor of Washington, D.C., and the governors of the Carolinas, Virginia, Georgia, and Maryland declared a state of emergency. President Donald Trump declared an emergency in North Carolina, allowing it to access federal funds. Six counties of North Carolina, namely Brunswick, Currituck, Dare, Hyde, New Hanover, and Onslow, were placed under mandatory evacuation on Sept. 10, while Durham, Johnston, Orange, Harnett, and Chatham counties were under high alert and possible evacuation. Franklin Township and the southern part of Sampson County
were placed under mandatory evacuation on Thursday, Sept. 13, providing them with much less time for evacuation. Eight counties of South Carolina along the state’s 187-mile coastline were placed under mandatory evacuation on Sept. 11. All the roads on I-26 and Route 501 were directed away from the coast. State government offices, including schools and medical facilities, were closed in 26 counties of South Carolina. In Virginia, mandatory evacuations were issued at around 8 a.m. on Sept. 11, for about 245,000 residents in a portion of Hampton Roads and the Eastern Shore area. Evacuations from coastal areas were proceeding normally on Interstate 40, U.S. Highway 70, U.S. Highway 74 and other routes. Around 10 million people resided in the storm’s path; around 250,000 people from Virginia, 750,000 people from North Carolina, and 500,000 people from South Carolina were issued evacuation orders. Raleigh officials used reverse-911 to notify its residents in low-lying areas about Florence and activated a non-emergency phone line for residents to get information about effects of Florence and issues that needed to be addressed. All the counties launched a text service to alert their residents and provide them updates about the hurricane. Across the three states, about 3,000 National Guard members and fourteen squads of State Highway Patrol troopers had been activated to assist with hurricane recovery. 126 shelters were opened across North Carolina and several more across South Carolina and Virginia. Approximately, 10,000, 40,000 and 4000 people were residing inside shelters in North Carolina, South Carolina, and Virginia, respectively.

Duke Energy, one of the major electrical utilities companies, predicted that 3 million people across the Carolinas would be affected by the power outage. More than 600,000 people were cut off from the power supply over the weekend, and 223,000 people were without power for the entire week. Duke had more than 20,000 workers working to restore power across North Carolina and South Carolina. It took more than two weeks, till Sept. 26, to fully restore power. Track vehicles in flooded areas replaced electric poles. Overall, power restoration was aided by over 40,000 workers.

Most of the air traffic operations in the areas affected by Florence were suspended. Many roads like I-40, I-95, US-17, US-70, etc. were closed due to flooding. President Donald Trump signed an emergency declaration for North Carolina, making federal emergency aid available to the state. Insurance payments and federal disaster aid helped in restoration. The Virginia National Guard announced plans to bring up to 1,500 soldiers, airmen, Virginia Defense Force members, and up to 6,000 personnel to aid in response operations of Hurricane Florence. The Diabetes Disaster Response Coalition created several resources to aid people with diabetes in areas affected by Florence.
11 User Manual

11.1 Installation Requirements

To configure the latest version of Spark, the installation and setup steps can be found at http://spark.apache.com/downloads.html.

11.1.1 Python Packages

1. NLTK
   The Natural Language Toolkit is a platform for building Python programs to work with human language data. To install NLTK, run

   ```
   pip install -U nltk
   ```

   The NLTK libraries used for the intent of this project were word_tokenize, pos_tag, stopwords, wordnet, punkt, averaged_perceptron_tagger, and chunked.

2. Gensim
   Gensim is a Python library for topic modeling. To install gensim, run

   ```
   pip install --upgrade gensim
   ```

3. SpaCy
   To install SpaCy, run:

   ```
   pip install -U spacy
   ```

4. NumPy
   Numpy is a package for scientific computing in Python. To install numpy, run

   ```
   pip install numpy
   ```

5. Matplotlib
   Matplotlib is a 2D plotting library which provides a simple API to generate plots in a variety of formats. To install matplotlib, run

   ```
   pip install -U matplotlib
   ```
6. **Python Rouge**
   Python Rouge is a Python wrapper to use ROUGE, which is a summary evaluation tool. To install pythonrouge, run
   
   ```
pip install git+https://github.com/tagucci/pythonrouge.git
   ```

7. **JusText**
   JusText is a tool to remove boilerplate content from HTML. To install jusText, run
   
   ```
pip install justext
   ```

8. **WordCloud**
   WordCloud helps us generate word clouds from text. To install wordcloud, run
   
   ```
pip install wordcloud
   ```

11.1.2 **TensorFlow**
   The easiest way to install TensorFlow is by using Python's pip package manager. The TensorFlow container can also be pulled using Docker with the command docker pull tensorflow/tensorflow. However, TensorFlow is only compatible with Python 3.5 and above. The next step would be to install Docker and deploy the Docker container by following the steps on the GitHub Repository for the course (https://github.com/xw0078/VT_Fall18_CS4984-CS5984).

11.1.3 **Git**
   To install the basic git tools on Linux -
   
   ```
sudo apt install git-all
   ```

11.1.4 **Mahout**
   Apache Mahout is an official Apache project and thus available from any of the Apache mirrors.
   
   Download the source code for Mahout using the command:
   
   ```
git clone https://github.com/apache/mahout.git mahout
   ```
Then, a few environment variables need to be edited. For Linux systems, in the /.

```
export MAHOUT_HOME=/path/to/mahout
```

### 11.2 Processing WARC and CDX files

Initially, the event folder should be put in the HDFS part of the cluster. The Archive Spark script is to be executed in order to convert the WARC file into JSON format. Make sure that the right path for JAVA_HOME directory is given.

```
$ export JAVA_HOME=/usr/java/jdk1.8.0_171/
```

![Figure 31: Running the ArchiveSpark scala script](image)

### 11.3 Using the scripts

It is important to ensure that the data files and Python scripts are in the same folder. If they are in different folders, the absolute paths should be given in the Python script. The scripts below should be executed in sequential manner as the output files generated by each script is the input for the following script.

#### 11.3.1 Text Cleaning using JusText

Run the script `jusText.clean.py` to remove boilerplate content and HTML tags from the data. Ensure that the JSON file `sentences_big.json` is in the same folder as the script file. The output file `cleaned_noise_removed.txt` will be generated.

```
python jusText(clean.py)
```
11.3.2 Data Preprocessing
Run the script ‘Preprocessing.py’ to tokenize the words, then remove the English language stopwords, stopwords specific to the dataset and then perform lemmatization based on the parts of speech of the word tokens. The input for the script is ‘cleaned_noise_removed.txt’ and it will generate ‘filtered_text.txt’ as output.

python Preprocessing.py

11.3.3 Data Exploration
Run the script ‘Exploration.py’ to extract the most frequent words, generate bigrams, identify named entities, and perform LDA topic modelling. The results will be visible in the terminal and at the same time, a bar graph and a word cloud plot will also be generated for the most frequent words.

python Exploration.py

11.3.4 Decision Rule Classification
First, run the script ‘remove_duplicates.py’ to remove all duplicate records from the data.

python remove_duplicates.py

Then, filter out the documents that do not contain the words ‘Irma’ or ‘irma’ using the script:

python classify.py

This will result in a cleaned output file ‘cleaned_dup_removed.txt’

11.3.5 Classification using Mahout
First, split the concatenated text file ‘cleaned_dup_removed.txt’ into individual files using the script:

python generate_individual_documents.py

Then, use the script ‘label_classify.py’ to split the manually labelled documents into the corresponding folders.

python label_classify.py
To perform the classification, first set the variable ‘WORK_DIR’ to the classify folder in the present working directory.

```
EXPORT WORK_DIR = pwd + '/classify'
```

Then, run the shell script ‘mahout_classification.sh’ to perform classification.

```
bash mahout_classification.sh
```

At this stage, we are left with only the relevant and significant documents which are used for generating the abstractive summary.

### 11.3.6 Clustering using Mahout k-means

To perform the classification, first set the variable ‘WORK_DIR’ to the individual folder in the present working directory.

```
EXPORT WORK_DIR = pwd + '/individual'
```

To perform clustering using Mahout k-means, run the script ‘mahout_clustering.sh’.

```
bash mahout_clustering.sh
```

### 11.4 Running the Pointer-Generator Network

We used the Python scripts from the original Pointer-Generator model to convert our data into the bin and vocab files. `Json_to_hash.py` converts all the documents in the input data in JSON file format separately into stories. In order to execute the script, the user must first navigate to the location of the cloned directory in the file system and run the following command on the command line.

```
python json_to_hash.py -f path/to/json/or/text/file -o path/to/stories/directory
```

The first argument indicates path to the input json or text file. The second argument indicates the output path to store the .story files.

In order to make the bin and vocab datafiles, the user must execute the `make_datafiles.py` file from the command line.
python make_datafiles.py <path/to/stories/directory>
<story type: train.bin, test.bin, or val.bin>

The first argument is the path to the stories generated by json_to_hash.py and second argument indicated what type of binary files (train, test or eval) should be generated from the input stories.

In order to run the pointer generator network, the run_summarization.py file must be executed from the command line

python run_summarization.py --mode=train --
data_path=/path/to/chunked/train_* --vocab_path=/path/to/vocab --log_root=/path/to/a/log/directory --exp_name=myexperiment

The first argument specifies the mode in which the pointer generator will run. The second argument specifies the path to the binary files corresponding to the mode. The third argument mentions the path to store the vocab file when running in train mode or path to load the vocab file from when running in decode mode. Finally, this will create a subdirectory of the user’s specified log_root called my experiment where all checkpoints and other data will be saved.

11.5 ROUGE Evaluation

ROUGE Evaluation is performed using the script ‘eval.py’ provided by the TA. To run the evaluation we need to provide 3 parameters to the script. The first, is the evaluation mode that is to be used -

- 1 - ROUGE\_para
- 2 - ROUGE\_sent
- 3 - cov\_entity

The second and third parameters are the folder paths for the golden standards and generated summaries respectively, if performing the ROUGE\_para evaluation. For the remaining evaluations, the absolute file paths are given instead.

- ROUGE\_para

  python eval.py -t 1 -g folder/path/golden_standard
  -p folder/path/abstractive_summary
- **ROUGE_sent**
  
  ```
  python eval.py -t 3 -g file/path/golden_standard -p file/path/abstractive_summary
  ```

- **cov_entity**
  
  ```
  python eval.py -t 3 -g file/path/golden_standard -p file/path/abstractive_summary
  ```
12 Developer’s Manual

12.1 File Inventory

1. jusText.clean.py script extracts sentences from the JSON file and removes noise like boilerplate from the extracted sentences. It takes a JSON file ‘sentences_big.json’ as input and outputs a text file in which each line is the content from a document and generates an output file ‘cleaned_noise_removed.txt’ that is used in the next step.

2. Preprocessing.py script tokenizes the sentences in each of the documents, normalizes all the words into lower case, and removes stop words which are a part of the NLTK stop word list as well as some given custom stop words. It reads the file ‘cleaned_noise_removed.txt’ as input and generates the output in file ‘filtered_text.txt’.

3. Exploration.py script generates the most frequent words, bigrams with their count, named entities, and topics using LDA topic modeling. It also displays the word cloud of the most frequent words in which the size of the words is proportional to their frequency of occurrence. It reads the file ‘filtered_text.txt’ as input and generates the list of most frequent words, bigrams, named entities, and LDA topics. It also creates a bar graph and word cloud plot of the most frequent words based on their number of occurrences.

4. remove_duplicates.py removes duplicate documents. That means if the same document occurs more than once, only one of the copies is kept, and the rest of them are eliminated. It reads the file ‘filtered_text.txt’ as input and stores the output in file ‘cleaned_dup_removed.txt’.

5. classify.py uses a word filter with words ‘Irma’ and ‘irma’ and removes all the documents, i.e., lines in this case. It reads the file ‘cleaned_dup_removed.txt’ as input and stores the output in file ‘cleaned_complete.txt’.

6. generate_individual_documents.py reads the filtered text file as input and then generates individual documents out of the concatenated file, so that these can be used for classification. It reads the file ‘cleaned_complete.txt’ as input and splits the concatenated document into single files with the index of the document as their file name and stores them in a folder named ‘individual’.

7. labels.txt contains the labels for 300 documents that were manually labelled to create the training set for classification.
8. label-classify.py will store the first 300 documents of the dataset into two separate folders namely ‘0’ and ‘1’, where 0 and 1 represent irrelevant and relevant documents, respectively.

9. mahout-classification.sh shell script will execute the mahout commands that are needed to perform Complement Naive Bayes Classification on the dataset. It will first generate a sequence file for the data, then convert it to TF-IDF vectors, and then use these vectors to train a classification model. Once the model is trained, it will run the model on the entire dataset to classify the documents.

10. mahout-clustering.sh shell script will perform Mahout’s k-means clustering on the dataset to cluster the documents. It uses the already generated TF-IDF vectors for the dataset and then performs clustering on those.

11. json-to-hash.py will read all the documents present in the preprocessed text/json file, create a separate ‘.story’ file for each document, and save them in the mentioned output directory. Each story file must have a unique name. They are named based on their index in the text/json file.

12. make-datafiles.py will tokenize all the stories created from json-to-hash.py using the Stanford Tokenizer (Stanford CoreNLP) and then convert the tokenized stories into binary files. The type of binary files created (train, eval, or test) will depend on the command line argument given to the code as mentioned in section 11.4.

13. run-summarization.py will take the mode in which the pointer-generator network should run as one of the arguments, and call appropriate functions from other Python files. In train mode, it will call the model file to create the PGN model, and that calls the other train functions. In eval mode, it will load the created model during the train process, and that calls the functions which perform required evaluations. In decode mode, it will again load the trained PGN model and perform the decoding.

14. model.py is where the pointer-generator network is created with various hyperparameters using TensorFlow.

15. data.py reads the train/eval/test data which are in the form of binary files and also the vocab file created from training the data, which is used in the decode mode to generate the summary.
16. batcher.py reads the input data, either of train, test, or eval data, and processes them into batches based on the hyperparameters. It performs the training and decoding. This file also encodes the data from each batch into the format suitable to the TensorFlow pointer-generator model. This code also have multi-threaded support for processing data into batches.

17. decode.py is where the summary is generated from the test data and the vocab file using the pointer-generator network created in the train mode, and evaluated in the evaluation mode.

18. beam_search.py provides functionality for the decode.py to perform beam search decoding by loading the checkpoints, events, and metadata generated from the training mode. It returns the best hypothesis found by the beam search mechanism for generating the summaries.

19. attention_decoder.py defines the decoder and initializes all parameters required by the decoder in the decode.py file and maintains the state of the decoder.

20. util.py contains some utility functions for configuring the host to run the TensorFlow model and logging the TensorFlow output.
13 Challenges Faced

- There was an excessive amount of noise present in the data corpus which includes boilerplate, stopwords, duplicate documents, or non-relevant documents. Preprocessing the data to eliminate the noise was very challenging and time consuming. We could reduce the data set to approximately 2 percent of the size of the JSON file generated initially.

- Hurricane Irma and Hurricane Harvey occurred in overlapping time frames which made it difficult to differentiate the documents between the two hurricanes. Also, the documents which were relevant to Hurricane Irma had personal information of people, and it was confusing whether to consider them to be relevant or non-relevant. We have chosen to consider them to be relevant because they do include significant information relating to the struggle during the hurricane and the losses people have faced.

- The decision of choosing a classification algorithm was quite challenging because of several issues as mentioned previously. We started with complicated approaches and continued to look into more complicated algorithms until we figured how a simple word filter that could save us all the effort and give us much better results.

- The number of documents was almost 7000. Generating summaries for each of those documents using the Pointer-Generator Network was inefficient. This is the reason we have combined multiple documents into a single document and given that as the input to the Pointer-Generator Network. In spite of doing that, we had to post-process the summaries generated by removing duplicate sentences and manually removing insignificant sentences. This was quite challenging, to reduce the summary down to two pages from two and a half pages.
14 Conclusion

The goal of this project was to generate an abstractive summary, given a data corpus of over 15,000 articles. Over the course of this project, we used various techniques and tools at different stages to get closer to our goal. At first, we started with a simple approach with naive assumptions. We worked on this iteratively and refined our solutions to get better results at each stage, meeting different goals along the way.

Preprocessing like noise removal, normalization and tokenization was done over the entire data to clean it so that the deep learning model could learn efficiently from the data. Classification and clustering was performed on the data to filter out the irrelevant documents after which there were approximately 7000 relevant documents. We then used the Pointer-Generator network to generate an abstractive summary of the data which provides all the necessary information about Hurricane Irma.
15 Future Work

As part of future work, we would like to create bi-grams and tri-grams to include cities and countries since our current script for post-processing the summary only capitalizes certain countries classified by the POS (Part of Speech) tagger as proper nouns. This left words like Dominican Republic converted incorrectly as Dominican republic. We also need to include abbreviations such as NHC (National Hurricane Center) to be capitalized in the script since the current scope of our script is not inclusive of that.

Since our attempt to classify the documents as relevant and irrelevant failed, we would like to try out a better classification technique so that we are able to further reduce the size of our dataset and remove as many irrelevant documents as possible.

Apart from that, we would also like to move further ahead with the clustering of documents, since we feel that the partial results were quite good and we had to stop just because we were not able to generate a complete dump of the results in time.

One of the other challenges we faced in post-processing was that the data listed relative days instead of absolute dates. This made it hard for us to generate a cohesive timeline of events using an automated script. In the future, we would like to extract the dates that the articles and tweets were published from the HTTP response of URL of the articles and automatically replace days of the week with absolute dates. This will enable us to answer questions like the path of the hurricane and the total death toll in a better way.
References


rzation, Springer, pp.3-13, 2012, Theory and Applications of Natural
Language Processing, 978-3-642-28569-1.

[13] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. 2014. Sequence to se-
quencelearning with neural networks. In Proceedings of the 27th Inter-
national Conference on Neural Information Processing Systems - Volume
2 (NIPS’14), Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence,
and K. Q. Weinberger (Eds.), Vol. 2. MIT Press, Cambridge, MA, USA,
3104-3112.

stractive Sentence Summarization with Attentive Recurrent Neural Net-
works. 93-98. 10.18653/v1/N16-1012.

A recurrent neural network based sequence model for extractive sum-
marization of documents. Association for the Advancement of Artificial
Intelligence.

Neural Attention Model for Abstractive Sentence Summarization. Com-

2016. Efficient Summarization with Read-Again and Copy Mechanism.

point: Summarization with Pointer-Generator Networks. arXiv preprint
arXiv:1704.04368


of summaries. Proceedings of the ACL Workshop: Text Summarization
Braches Out 2004. 10.

proach to Preprocessing Text Data.” Retrieved from
https://www.kdnuggets.com/2017/12/general-approach-preprocessing-
text-data.html

61