

DETECTION OF PAST NATURAL CATASTROPHE EVENTS THROUGH HISTORICAL AND CURRENT NEWS ARTICLES

Albert Galang

Bachelor of Engineering
Software Engineering



School of Engineering
Macquarie University

September 02, 2019

Supervisor: Dr. Tahiry Rabehaja

ACKNOWLEDGMENTS

I would like to acknowledge my supervisor Dr. Tahiry Rabehaja for his absolutely spectacular guidance throughout the undertaking of this undergraduate thesis. I would also like to thank Macquarie University for giving me the opportunity to undertake this project and supplying me with the resources to do so.

STATEMENT OF CANDIDATE

I, Albert Galang, declare that this report, submitted as part of the requirement for the award of Bachelor of Engineering in the School of Engineering, Macquarie University, is entirely my own work unless otherwise referenced or acknowledged. This document has not been submitted for qualification or assessment at any academic institution.

Student's Name: Albert Galang

Student's Signature: Albert Galang (electronic)

Date: September 06, 2019

ABSTRACT

Throughout the past century Australia has experienced a plethora of natural disasters, but many of those life changing events that date before the 1960's have gone unrecorded. There exists multitudes of data that prove the occurrence of these unrecorded events, yet for the purpose of catastrophe modelling, there has been minimal effort in the usage of such data to pinpoint when these life changing disasters had taken place. The purpose of this research is to exploit the aforementioned data, using modern techniques to extract information that will be used as the foundation of a model that is able to detect the presence of natural disasters. It is with high hopes that this project will provide the groundwork for future research into catastrophe modelling, and that it will serve as an example of one of the many ways in which we can process and analyse the veritable deluge of information that exists online.

Contents

Acknowledgments	iii
Abstract	vii
Table of Contents	ix
List of Figures	xiii
List of Tables	xv
1 Introduction	1
1.1 Project Objective	1
1.2 Outcomes	2
1.3 Project Plan	4
1.3.1 Scope	4
1.3.2 Timeline	4
1.3.3 Cost	6
1.4 Document Structure	6
2 Background and Related Work	7
2.1 Information Retrieval and Processing	7

2.1.1	Transforming Unstructured Text into Structured Text	7
2.2	Machine Learning Approaches to Event Detection	10
2.3	Sourcing Data	12
3	Data Retrieval and Preprocessing	15
3.1	Data Retrieval	17
3.2	Preprocessing Data	21
3.2.1	Cleaning	21
3.2.2	Tokenization	22
3.2.3	Removal of Stop Words	22
3.2.4	Normalizing Text	23
3.2.5	Calculating Frequency	23
3.3	Storing Information	25
3.4	Filtering Function	27
4	Feature Selection	29
4.1	The Sliding Window Method	29
4.2	Article Frequency as a Feature	33
4.3	Word Frequency as a Feature	35
4.3.1	Determining the Input Type	35
4.3.2	Determining the Important Terms	37
5	Data Modelling and Evaluation	39
5.1	Dataset and Model Construction	39
5.2	Model Evaluation	43
5.2.1	Comparing Models Trained on Undersampled Data versus All Data	44

5.2.2	Altering Features based on Feature Importance	44
5.2.3	Choosing the Machine Learning Algorithm	49
6	Maintenance and Testing	51
6.1	Documentation	51
6.2	Version Control	52
6.3	Testing	52
7	Discussion	53
8	Conclusions	57
9	Future Work	59
9.1	Potential Improvements	59
9.2	Final Words	60
10	Abbreviations	61
A	Consultation Attendance Form	63
B	Code	65
B.1	Trove API Data Retrieval Functions	65
B.2	Article Cleaning, Preprocessing, and Storage	67
B.3	MongoDB Retrieval and Dataset Construction	73
B.4	Feature Selection	81
B.5	Model Construction	93
B.6	Detection Functions	96

Bibliography

100

List of Figures

1.1	Finalized RF Machine Learning Model Tested on Data 1940 - 1949	3
1.2	High-Level System Flowchart	3
1.3	Project Plan Gantt Chart	5
3.1	Text Preprocessing Flowchart	16
4.1	Text Preprocessing Flowchart	30
4.2	Proximity Range versus Weighted Correlation Values	32
4.3	One and Five Star Frequencies Against Event Occurrence 1930 - 1939 . . .	34
4.4	Average Star Ranking Against Event Occurrence	35
4.5	Frequency Types against Event Occurrence September 1902 - December 1902	36
5.1	Model Construction and Evaluation Flowchart	40
5.2	Regression Dataset with Undersampling 1940 - 1949	42
5.3	Regression Dataset without Undersampling 1940 - 1949	42
5.4	RF Regression Model Predictions Trained on Undersampled Data 1910 - 1919, Tested on Data 1940 - 1949	45
5.5	RF Regression Model Predictions Trained on All Data 1910 - 1919, Tested on Data 1940 - 1949	45
5.6	RF Regression Non-Article Frequency Model Predictions Trained on All Data 1910 - 1919, Tested on Data 1940 - 1949	48

5.7	RF Regression Average Star Frequency Model Predictions Trained on All Data 1910 - 1919, Tested on Data 1940 - 1949	48
-----	---	----

List of Tables

3.1	Preprocessing Text	21
3.2	Articles before and after filtering	27
4.1	Example Dataset	31
4.2	Correlation Values of Article Frequencies of Rankings	34
4.3	Correlation Values of the term ‘Cyclone’ Various Word Frequency Input Types	37
4.4	Highest scoring tf-idf terms within 4 days of event occurrences 1930 - 1939	38
5.1	perilAUS Number of listed tropical cyclone events every ten years 1900 - 1949	40
5.2	Feature Importance returned by model trained on data from 1910 - 1919 and tested on data from 1940 - 1949	47
5.3	Performance Metrics of RF Regression Models Trained on All Data 1910 - 1919 and Tested on Data 1940 - 1949	47
5.4	Performance Metrics of Different ML Regression Models Trained on All Data 1910 - 1919 and Tested on Data 1940 - 1949	49
7.1	Predicted Events with High Confidence Intervals and Corresponding Arti- cle IDs	55

Chapter 1

Introduction

Occurrences of natural disasters are no doubt some of the most important points in history, yet there is a distinct lack of records pertaining to these events when investigating the years before the 1960's [1]. Many of the recorded natural disasters during this early time period had only been recorded due to their subsequent high insured losses. Consequently, there is a lack of recorded disasters that still had relatively heavy impacts on those affected, but were uninsured. These missing pieces of information are not only historically significant to those who had been affected by these life changing events, but also to those who study catastrophe modelling and are interested in verifying the completeness of their own data sets.

Ignorance towards these past disasters may have been acceptable one or two decades ago, but in the modern era there exists multiple techniques that allow us to salvage those that were lost or forgotten. These techniques range from machine learning algorithms to natural language processing kits. Although these exist, there has been little effort made to pursue the reclamation of these forgotten disasters.

1.1 Project Objective

The primary objective of this project is to be able to accurately detect the presence of past natural catastrophes through the extraction and processing of information retrieved from already existing historical articles. This is to be delivered through a functioning application that accepts historical data within a time frame as the input, and returns the temporal locations of natural disasters that occurred within the said time period. The bare minimum requirement is that the output should return a confidence interval of the predicted natural disaster, and the date in which the prediction occurs.

In order to achieve this goal, copious amounts of research into relevant machine learning algorithms and natural language processing tools will be necessary. This project aims to research and understand these techniques through an intensive literature review. Then to use this research and apply it to the already existing abundance of data present on the internet, reclaiming the neglected records of past natural disasters.

The project is to be divided into various subgoals, each a prerequisite for the completion of the final application which are outlined in section 1.3.1.

1.2 Outcomes

The system must be able to correctly predict the dates of already listed events. This confirms the validity of the systems predictions, and gives credence to other predictions made for events that are unlisted. These unlisted event predictions are given confidence intervals depending on how confident the system is in it being an actual event. They are manually investigated and then classified into either an actual unlisted event, or a false positive.

These goals are achieved with varying degrees of success upon project completion as evidenced by the predictions of our finalized machine learning model shown in fig. 1.1 for the years 1940 - 1949. This model successfully predicts 16 out of 20 listed events, and through manual investigation of predictions with high confidence intervals ($>90\%$) we are able to find the presence of an unlisted actual event. This validates our project goals. The processes employed to achieve these results are described in-depth throughout this report.

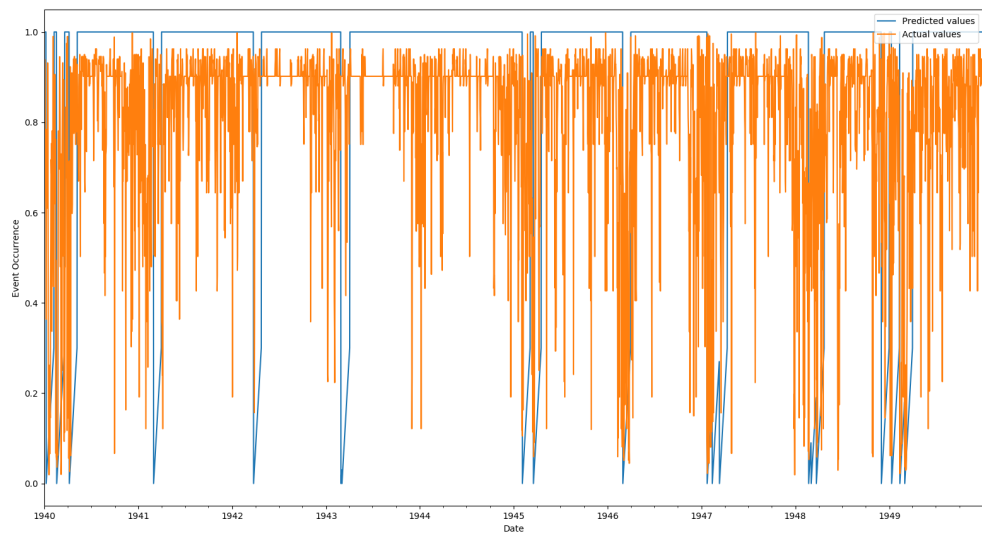


Figure 1.1: Finalized RF Machine Learning Model Tested on Data 1940 - 1949

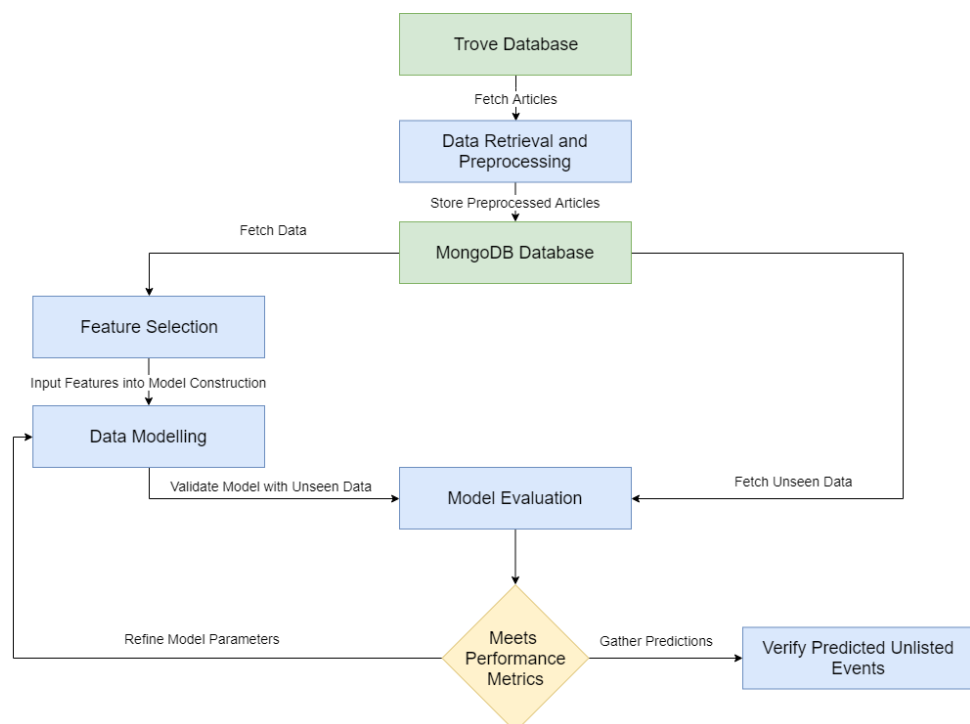


Figure 1.2: High-Level System Flowchart

1.3 Project Plan

It is important to define the strategy necessary to undertake the project, as it will assist in maintaining an organized effort. This section outlines the overall scope of the project, alongside the predicted timeline, cost, requirements and structure. The overall high-level flowchart for the proposed system is showcased in fig. 1.2.

1.3.1 Scope

Extensive background research is necessary before and during the undertaking of the project, as it will influence the amount of time given to various stages, and will allow us to validate the completeness of the final project. As mentioned in Section 1.1, the application development is to be split into various subgoals that are to be achieved within the specified time frames. These essential objectives are as follows:

- Data acquisition
- Feature selection
- Data modelling
- Model evaluation

Each of these requires research and experimentation with in order to succeed in constructing the final application. Each has been allocated varying timeframes over the thirteen week period of the project in accordance to how difficult it will be to complete. Non-essential objectives also include a fleshed out graphical user interface (GUI), acceptance testing, and extension to other events. These non-essential objectives are to be completed if the time frame allows it once the essential objectives are completed, but are unnecessary for the application to function.

1.3.2 Timeline

As shown in figure 1.3, each of the aforementioned goals and subgoals are plotted over a thirteen week timespan. This Gantt chart was created during the planning stages of the project, and therefore is only a prediction of how long each goal should have taken to complete. It can be seen that the goals of data acquisition, data modelling and model evaluation were judged to be of equal difficulty, and thus similar amounts of time were

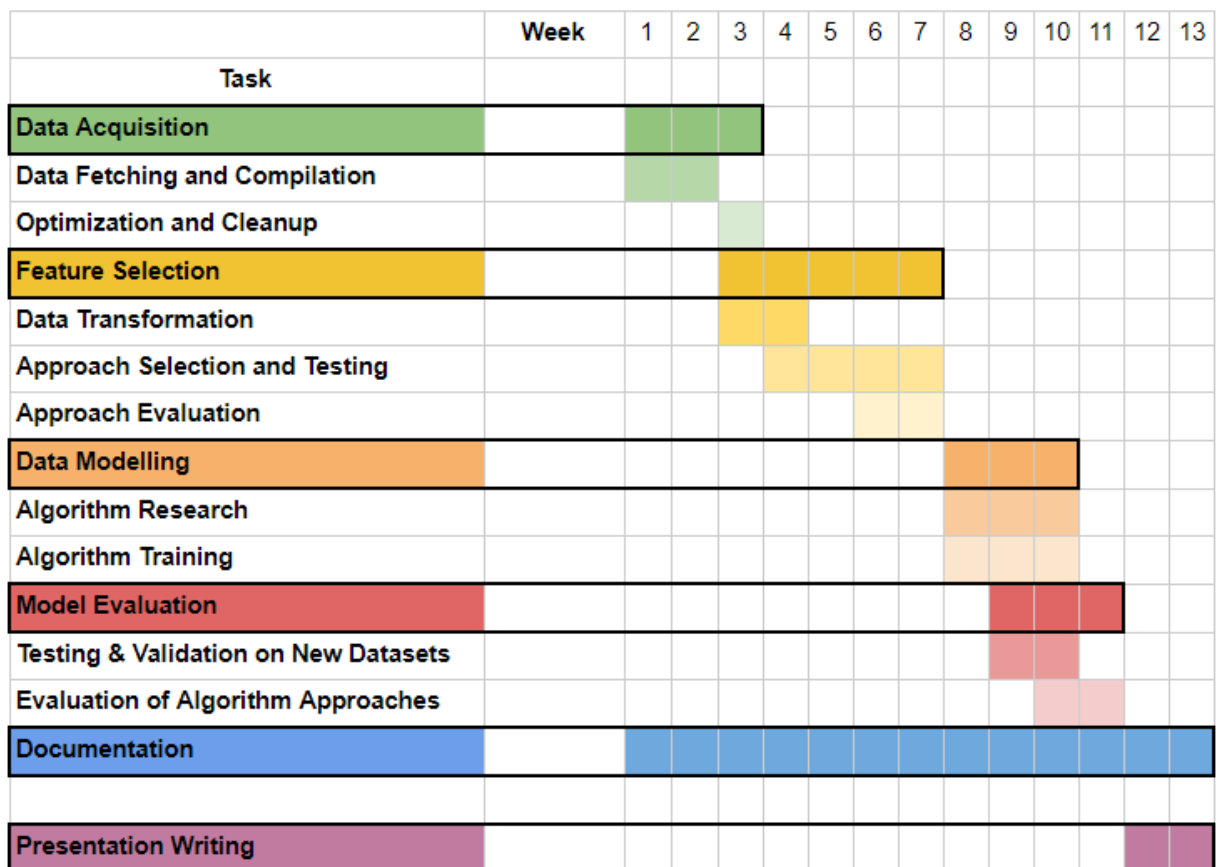


Figure 1.3: Project Plan Gantt Chart

allocated to those goals. From research, it can be seen that feature selection would be the most labour intensive goal to achieve, and thus was allocated much more time.

During actual execution of the project, things did not go as smoothly as planned. The stage of data acquisition had been completed in a much shorter time frame than expected, thus allowing more time for feature selection. There had also been much overlap between data modelling, model evaluation, and feature selection than expected, as it was necessary to verify the correctness of input features. It had also been necessary to go back to the data transformation stage at various points to tweak and restructure the inputted data for feature selection.

Retrospection indicates that although having an organized and planned out timeline is necessary for timely completion of the project, the timeline should not be seen as a fixed strategy but instead as a guideline to follow. This is because complications will inevitably arise, causing some goals to take longer or shorter periods of time than expected.

1.3.3 Cost

In terms of monetary costs, although there was access to a AUD \$400 budget for the project, both plans and current execution indicate that utilising it will not be necessary. This is mainly due to the usage of free resources available online, these include natural language processing (NLP) libraries such as the Natural Language Toolkit (NLTK) and machine learning libraries such as NumPy. One exception to this is the Python integrated development environment (IDE) PyCharm, which if not for the available free student licence, would have cost USD \$200.

1.4 Document Structure

This document aims to describe various aspects of the project that contribute to the final solution. The document can be seen as three major divisions of work.

The first of which are chapters 1 and 2 which contain the necessary background knowledge and plans required to construct the solution. The second division include chapters 3, 4, and 5, these contain the main procedures experimented with to generate results and a functioning solution. The third and final division are chapters 6, 7, 8, and 9. These chapters encompass the aftermath of the project, in which we will discuss encountered issues, results and future work.

The appendices provide the code used to implement the solution, advisor progress history, and requirements.

Chapter 2

Background and Related Work

Research and revision of literature related to the issue is vital to project completion. Background research allows us to compare and contrast other approaches and solutions to the same or similar problems. We can construct our own approaches and methodology by examining and incorporating what approaches worked for other researchers. The literature review also allows us to construct a thorough foundation of the techniques and libraries that will be necessary for the solution to function.

2.1 Information Retrieval and Processing

“Information retrieval is finding material of an unstructured nature that satisfies an information need from within large collections” [2]. Before gathering the features required to build a detection model is possible, we require an abundance of data from which we can extract these features. This data will be populated by unstructured text in the form of news articles fetched from historical news aggregators. The project will therefore require the ability to organize the retrieval and analysis of tens of thousands of articles. It is thus of utmost importance that we review not just the methods in which to accomplish this, but also how we can optimize said methods. We first look towards what is required to process unstructured textual data into data that we are able to manipulate and extract features from.

2.1.1 Transforming Unstructured Text into Structured Text

Developing an application that can analyse and understand unstructured text is an arduous and complex task. This is because there exists multiple ambiguities in natural

language that allows the possibility of interpreting the same sentence in many different ways. These include morphological, lexical, referential and syntactic ambiguity [3, 4]. Research indicates that there are multiple approaches to transforming this unstructured textual data into structured data. Broda et al. (2013) provides a regular expression language after processing textual data such that it can be directly queried [5]. Krishnamurthy et al. (2009) describes their application which leverages database concepts to enable complex information extraction, the researchers succeed to reduce the running time of complex extraction [6]. It may be unfeasible within the allocated time frame to utilise complex approaches such as the aforementioned when attempting to solve the problem at hand. In order to simplify the overall process, we can instead aim to extract features that require simple processing methods such as term usage and word frequency. This requires a thorough understanding of some of the foundational concepts of natural language processing (NLP). Namely, the techniques of tokenization, stemming, lemmatization and stop word removal [7].

Tokenization

It is simply impossible to analyse natural language without the basic segregation of text into their basic units of words [8]. This is the process of tokenization and it is one of the first and most essential preprocessing steps of NLP. Where sentences are broken down into separate items, known as tokens. This process is absolutely necessary because it allows the succeeding functions to process data in pieces, rather than as a whole. It would otherwise be impossible to stem or lemmatize a whole block of unstructured text. Haoda et al. (2009) outlines a few important issues that may be encountered during tokenization, these issues are mostly due to the inability to recognize sentence boundaries due to ambiguous punctuation. For example, whether \$3.0 should be parsed as `[$|3|.|0|` or as `[$3.0|`. Fortunately, occurrences such as these are rare relative to the amount of information that will serve as the input for the application. They should have little effect on the overall outcome. Once unstructured text is tokenized, the next stage normalization will occur.

Text Normalization

Normalization is the process of “replacing several forms by a single canonical form” [9]. Both stemming and lemmatization are processes that achieve this normalization based on the morphology of words. The only difference between the two is the result. Stemming strips the suffix off the word and returns the result as a stem. This often leads to results that are not actual words. For example stemming “deranged” would return the stem “derang”. Whereas lemmatization will return what is called a lemma, which is an actual dictionary word. Lemmatization as Kimmo et al. (2005) defines, is the process when the

inflected word form and its base form are related with each other with an algorithm [10]. When comparing the two, we can refer to Liu et al. (2012) who reasons that lemmas show more advantages in document clustering and information extraction due to stemming often returning ambiguous and truncated stems [11]. The reason normalization of the tokenized text is a necessary step in the creation of any dataset is because it serves as a form of dimensionality reduction, and will thus improve algorithm efficiency [12]. In essence, performing normalization of the data reduces the range of variables, in this case words in the form of tokens. There are thus fewer dimensions of information required to sort through.

Stop Word Removal

The last concept to understand is that of stop words, which are non information-bearing words. Stop words are equivalent to the noise in a collection of data, they are redundant and useless terms we wish to filter when processing text. Common examples of stop words are “the” and “a”. Various premade lists of stop words exist online, removing the need to construct one from nothing. Baradad and Mugabushaka (2015) argue that the usage of corpus specific stop words might help in textual analysis [13]. This makes sense in that if a corpus of data were to be focused around a specific topic, then there would be higher frequencies of words that are not usually seen as stop words, yet are still redundant in the context of the corpus topic. The reason we filter these words out before analysing data is the same reason we normalize the tokenized text: there are fewer words to analyse and therefore the time taken for the algorithm to run is lowered. Removal of stop words is a form of dimensionality reduction aimed to increase the efficiency of the overall algorithm.

It is important to understand these concepts as they serve as the foundation for information retrieval and processing, and are thus necessary to achieve our goals. From there, we can also begin to adapt these algorithms to enhance the efficiency of our application. For example, sufficient understanding of the concept of stop word removal allows us to tailor our own stop word list such that it is better suited to the context of natural catastrophe detection. We may be able to utilise external libraries to execute these methods, but such an action would be fruitless without a solid understanding of how they work. Ample comprehension of the various techniques and methods in which we can go about information retrieval and processing, allows us to select the best methods overall.

2.2 Machine Learning Approaches to Event Detection

Event detection is the major focus of this research project, and is thus one of the most important concepts to perform research on. Machine learning is closely intertwined with event detection, as it serves as a method of information extraction necessary to build a model that detects events. Event detection is the process of identifying various parameters and characteristics of an event and implementing those over a range of data to find whether such an event has occurred within that range. In the case of this research project, we utilise event detection techniques to discover natural catastrophe events through a specified time frame of data gathered from historical articles. Preliminary research indicates that there are a wide variety of scholarly articles published on event detection, each applying a multitude of different machine learning approaches to yield varying amounts of success. Through reviewing of these scholarly articles, we aim to identify the most successful and suitable approaches that may be applied to our own project.

Comparing Approaches

Valero et. al (2009) aim to enhance the acquisition process of natural disaster data through reducing the time taken to quantify the impact of the natural phenomenon [14]. The researchers utilise a supervised machine learning approach in order to automatically extract information from natural disaster news reports. They attempted to extract terms relating to the temporal and geographic locations of the disaster, and the quantity of losses induced by said disaster. Our project aims to achieve a similar goal; to extract relevant features indicative of a natural catastrophe but instead of predicting the disaster magnitude, we aim to predict the likelihood of an occurrence. Valero et. al evaluated their findings through two different weighting schemes, Boolean and term frequency, then applied three different machine learning algorithms, namely Support Vector Machine (SVM), Naive Bayes, and C4.5. Results indicated that the combination of boolean weighting and SVMs outperformed the results of the other two learning algorithms. Ireson et al. (2005) supports this conclusion, when performing comparisons between rule induction, SVM and conditional random field (CRF) for information extraction [15]. Another approach is outlined by Zhou. et al (2017) when attempting to construct a systematic approach to harvesting, processing, and analysing social media data [16]. The researchers classify their dataset of tokens processed from Twitter feeds and their results suggest that the random forest (RF) algorithm provides the best accuracy score in comparison to k-nearest neighbour, Naive Bayes and SVM.

Inspection of the above examples allows us to gather many different and useful techniques for our own project. Firstly, the terms in which the researchers used to identify

their events are of great benefit to us, as they are similarly attempting to identify large events. These terms relate to those such as fatality rates, disaster magnitudes, economic and infrastructural damages. We can use these as a foundation when commencing the feature selection stage, as they may prove to be valuable indicators of event occurrence. Another advantage granted from the revision of these papers are the researcher's evaluations of various machine learning approaches. Not only do they provide insight into valuable measures in which we can benchmark our own machine learning models against, but also identify the machine learning models in which they had the most success with. It is interesting to note that although both studies are on event detection, there is a conflicting opinion between which machine learning algorithm is best suited. This could be attributed to the differing datasets and feature inputs of their models. This serves as a reminder that there is no definitive superior algorithm. The only way which we are able to discover what provides the best results is by experimentation with our own dataset.

A Modern Approach

Sayyadi et al. (2019) utilises a label based clustering approach as a method for automatic news event detection [17]. Sayyadi bases the method on the assumption that there can be several titles that identify an event uniquely and that every article relating to that event contains at least one of these titles. Events are found through similarity comparison in each individual cluster, those with high internal similarity labeled as an event. In order to compensate for events that sprawl over several days, and therefore over several clusters, the researchers also merge clusters if there is a similarity between consecutive days. The researchers generate titles through the JMontylingua module, and extract events using the aforementioned clustering approach. Different from the previously mentioned approaches, the labels are generated on a case-by-case basis, in that they are different for each event. Sayyadi et al. provides a fresh and modern perspective on tackling the challenge that is event detection through news articles, utilising a unique label-based model that identify events within clusters of articles. This provides another avenue which we are able to explore when implementing our own detection models.

Anomaly Detection

Natural catastrophe detection over a period of time through historical news articles is fundamentally a form of anomaly detection over sequential data. Since there are a limited amount of scholarly articles on natural disaster detection through news articles, we can expand our scope of research by looking towards other anomaly detection challenges. Lane and Brodley (1997) discuss a few key concepts when applying machine learning algorithms to anomaly detection of user profiles [18]. In particular, they discuss the classification threshold used is a static number, but concede that using a single setting was crude when

their other parameters were subject to change. This highlights one of the core issues that we must tackle when implementing our own designs, which is how we classify data as either an anomaly (natural disaster occurrence) or normal. This issue is further amplified by the fact that our dataset has a high variation of articles per year, which may not be due to event occurrences. Dietterich (2002) outlines some of the leading methods for solving sequential supervised learning problems [19]. The methods are namely the sliding window, the recurrent sliding window, hidden Markov Model (HMM), CRF and graph transformer networks (GTN). Dietterich highlights the drawbacks and advantages of each method, which will be of use when choosing how our own model will be designed.

Event detection remains as one of the most integral components of this project, and thus thorough revision into the surrounding literature was necessary. From this revision, we are able to grasp the varying types of machine learning approaches to event detection and use these later on to build the foundation of our own project.

2.3 Sourcing Data

In order to accomplish the goals set as mentioned in chapter 1, we require copious amounts of data to populate our model. There are two sets of data we require, firstly the corpus of news articles themselves, and secondly the list of natural disasters that have occurred in Australia alongside their dates.

Trove

We will be utilising the Trove available non-commercial, free to use application program interface (API) [20]. Trove is an Australian online news and database aggregator, and stores an extensive amount of digitized articles dating back to 1800. The reason we are using Trove is because obtaining and utilising the API key is a relatively simple process, and that there is an abundance of information we are able to obtain from it. The biggest issue encountered with Trove is their Optical Character Recognition (OCR) software misidentifying letters when scanning digital images of newspapers [21]. This is attributed to a multitude of factors such as source quality or image type. This is an issue because there are often nonsensical words in our data that are formed due to incorrect letter recognition, thus contaminating the data. There is nothing we can do at the moment to remedy this issue, we must instead attempt to remove the most common nonsensical words and ignore the rest. There is also a 100 API calls per minute quota instantiated for free users, which should not be too much of an issue. In the context of our own project, we are mostly interested in articles beginning from the year 1900, this is because that is when data is most abundant for both our data sources.

PerilAUS

PerilAUS has provided us with a list of approximately 17,000 natural disasters and their dates dating back to 1800. The list is set in a simple database format that has two columns, 'Peril' and 'Date'. Peril is the type of disaster e.g 'Flood', 'Tornado', 'Bushfire', etc. The only issue when parsing this file is that some of the entry dates are incomplete, either the month, day, or hour of occurrence is missing. We can disregard the hour since it is currently irrelevant in the context of our project, but we are faced with little choice but to filter out those entries that are missing month or day information. This is because they are vital in training our models to detect the disaster itself.

Chapter 3

Data Retrieval and Preprocessing

In this chapter we will detail the functions and libraries used to retrieve, preprocess, and store information from our data sources (Trove and PerilAUS). This is important as the resulting database will be filled with structured article text and event data that we are able to analyse and build a model from. The following experimentation is done with the goal of detecting cyclone events in Australia, thus all of our searches and datasets contain information regarding cyclones.

Necessary Tools

There are a few essential tools worth mentioning and libraries we utilise to simplify and optimize the endeavour of processing information. These are:

- PyCharm - The IDE we use to write our Python code
- NLTK - We use multiple functions from this library to clean our text
- MongoDB - A database storage program

We use PyCharm as our IDE to streamline the coding process. It is much easier to install the modules we require for our functions, and also has integrated Git functionality simplifying version control. NLTK has inbuilt tokenization, lemmatization, and stop word removal functions that have already been optimized. We can use these to circumvent the process of building our own, thus giving ourselves more time to focus on building our prediction model. MongoDB will be how we achieve local storage and data access. Storing our preprocessed articles removes the need to restart the long process of collection and cleaning every time we want to experiment with our data.

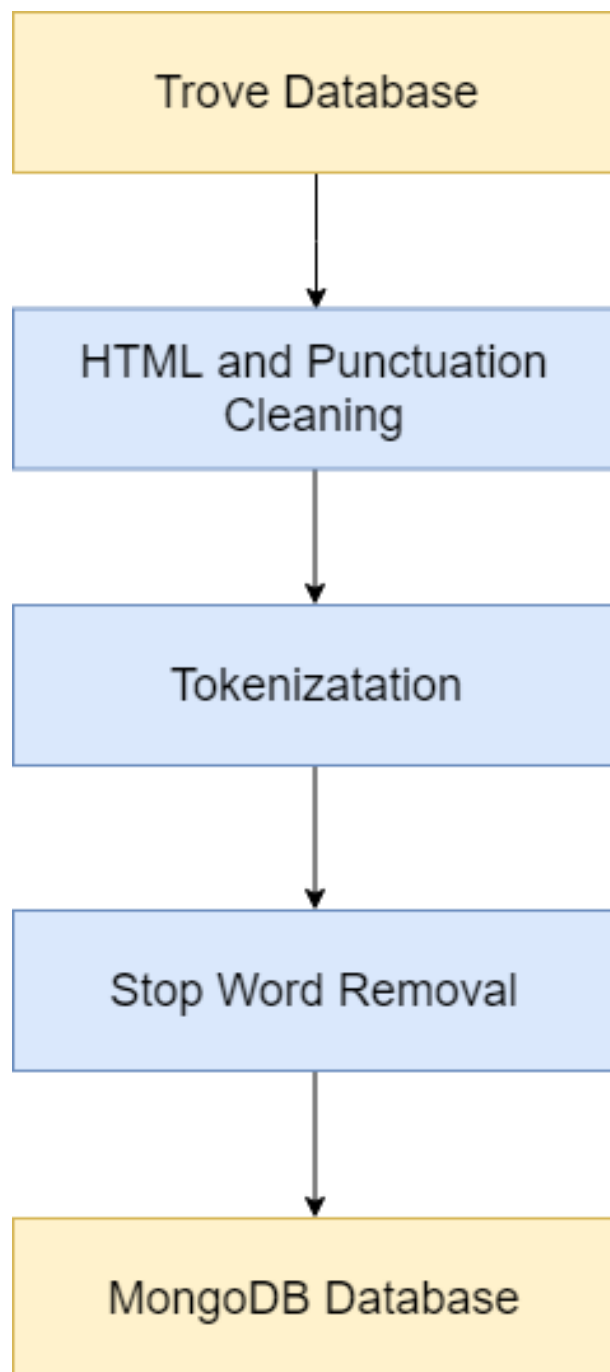


Figure 3.1: Text Preprocessing Flowchart

3.1 Data Retrieval

Our data comes in the form of an offline spreadsheet of peril events (PerilAUS) and an online database of historical articles (Trove). Both are relatively simple to fetch data from, but there are some issues that are prone to occur during these processes. This section seeks to outline these processes and the methods used to fix the issues.

Trove Retrieval

Requesting data from Trove using the provided API is quite simple, we only need to follow the provided guidelines and input our search parameters to return results. There are a few recurring errors that we are unable to remedy as they originate from the Trove servers, and not our implementation. The most important one to catch is when we receive a `HTTPError` from the server. We can attempt to remedy this by instantiating ourselves a while loop which iterates with each subsequent request sent to the server. Experimentation found that an immediate attempt to reconnect always resulted in another `HTTPError`, thus we wait a few seconds before we try again. If we fail to get a response ten times, the code returns an error. Fortunately this has never occurred, as a response is usually seen within three attempts.

```
1 @on_exception(expo, RateLimitException, max_time=60)
2 @limits(calls=100, period=60)
3 def call_api(url):
4     """Requests response from URL. Complies with call limit.
5
6     :param String url: valid encoded url request
7     :return: response
8     :rtype: object
9     """
10    attempts = 0
11    while attempts < 10:
12        try:
13            response = urllib.request.urlopen(url)
14            return response
15        except HTTPError as e:
16            print("Encountered Error: ", e)
17            print("Attempting to reconnect...")
18            sleep(6)
19            attempts += 1
20
21    response = urllib.request.urlopen(url)
22    if response.getcode() != 200:
23        raise Exception('API response: {}'.format(response.getcode()))
24    return response
```

Listing 3.1: API Call Function

The response to a successful request is a list of results with their information in the JavaScript Object Notation (JSON) format. We specifically request this format as we are able to easily read and manipulate it through Python. Below is the function that constructs our request to the Trove server, here we can specify our parameters. We are currently using the term ‘Cyclone’ as our search keyword, we request the maximum amount of results from the server that can be shown in a single response, and limit the search to Australia.

```

1 def trove_api_request(trove_key, zone, category, search_terms, min_year,
2   max_year, s, n):
3     """Makes a search request to the Trove API with the provided
4     search_terms and returns the selected page of results in JSON format
5
6     :param trove_key:
7     :param zone:
8     :param category:
9     :param search_terms:
10    :param min_year:
11    :param max_year:
12    :param s:
13    :param n:
14    :return:
15    """
16
17    search_terms_url = url_encode(search_terms)
18
19    request = "https://api.trove.nla.gov.au/v2/result?key=" + trove_key
20    + \
21        "&zone=" + zone + \
22        "&q=" + search_terms_url + \
23        "%20date%3A%5B" + str(min_year) + \
24        "%20TO%20" + str(max_year) + \
25        "%5D&encoding=json&s=" + str(s) + \
26        "&n=" + str(n) + \
27        "&relevel=full&include=articletext&l-australian&l-
28    category=" + category
29
30    # print(request)
31    response = call_api(request)
32    response_content = response.read()
33    response_json = json.loads(response_content)
34    return response_json

```

Listing 3.2: API Request Construction Function

Construction of the aforementioned two functions is all that is required to successfully

request and receive data from Trove. These functions are called repeatedly throughout the preprocessing stage as there are thousands of results for articles in a single year. There is an unfortunate constraint that there is only a maximum of 100 results per response page [22]. This means that if we wanted to process 3000 results, we would need to send a request at least 30 times.

This leads into one of the biggest issues plaguing the success of this stage: faulty response pages. For example we are attempting to process 2000 results, we must first send a request that returns a results page with 100 results. We use a parameter called 'nextStart', which is often a key of strings and integers from that response to get the next page of results. The issue occurs here, where the 'nextStart' key sometimes directs the application to a faulty page of empty results, even if there are still results that have not been shown. Requesting again with the same key only results in the same empty page of results, and the only remedy we have successfully implemented so far is to start the call over again from the start. Fortunately, we are able to save the already processed pages of results and attempt to access that page again with a different key from the new cycle.

The most frustrating component of this issue is that it occurs both randomly and at fixed page numbers. For example, there are approximately 5000 results regarding cyclones from 1905, and are thus 50 pages of results the application needs to process. The 43rd page will always be a faulty page, regardless of how many times the function tries again from the start. The issue randomly occurs between the 1st and 43rd page also, leading to an infinite cycle of calls. This is remedied by taking note of the highest page reached and skipping if the issue occurs at that page within a specified amount of times. We attempt to minimize this issue by always using the maximum number of results per page (100), and querying results in chunks of single years, rather than in a single query. For example, if we were to grab articles from 1900 to 1905, we first query 1901, and then begin a new one for 1902. Unfortunately, this method of exiting means that there will always be a portion of articles unanalysed and therefore not in our dataset for that year.

The final expected result from this retrieval is a list of dictionaries, each dictionary representing a single article. This dictionary should contain the following information:

- Date
- Processed Text
- Word Count
- Word Frequency
- Word Frequency vs. Word Count

PerilAUS Retrieval

The PerilAUS data comes in the format of a comma-separated values (CSV) file, thus it is simple to iterate and store the results into our own database. The code below shows the function we use.

```

1 def read_csv(filename):
2     """Reads csv file, not inputting any with missing date information
3     up to day i.e YYYY/MM/DD, hour is not recorded
4
5     :param filename:
6     :return peril_list: A list of dictionaries with peril and date in
7     datetime format
8     """
9     reader = csv.DictReader(open(filename))
10    peril_list = []
11
12    for row in reader:
13        if row['Start Year'] == 'NULL' or row['Start Year'] == '0' or \
14            row['Start Month'] == 'NULL' or row['Start Month'] == '0'
15        ' or \
16            row['Start Day'] == 'NULL' or row['Start Day'] == '0':
17            continue
18
19        peril = row['Peril Type']
20        date = row['Start Year'] + '-' + row['Start Month'] + '-' + row[
21        'Start Day']
22        try:
23            date = datetime.strptime(date, '%Y-%m-%d')
24            peril_dict = {'peril': peril,
25                          'date': date}
26            peril_list.append(peril_dict)
27        except ValueError as e:
28            print(e)
29            continue
30
31    return peril_list

```

Listing 3.3: PerilAUS CSV Reader

The single issue encountered during this process was outlined earlier in section 2.3 and is dealt with in the above code. This issue is that of an invalid date value in the date column. In response to this, we catch `ValueErrors` when attempting to retrieve and transform values in the date column into actual datetime values. These `ValueErrors` often occur when a null value or otherwise invalid date value is parsed. In this case we skip over this peril, as we require a full date in order to succeed in building an event detection model further on.

Sample Text	<html><body><p>Here's some generous sample text! </p></body></html>
HTML Removed	Here's some generous sample text!
Punctuation Removed	Here s some generous sample text
Tokenization	['here', 's', 'some', 'generous', 'sample', 'text']
Stop Word Removal	['generous', 'sample', 'text']
Stemming	['gener', 'sampl', 'text']
Lemmatization	['generous', 'sample', 'text']

Table 3.1: Preprocessing Text

3.2 Preprocessing Data

In this section we utilise the concepts researched in section 2, that of natural language processing techniques. We rely heavily on the usage of the NLTK library functions in order to keep our functions optimized, although there are also additional exceptions we add on to cater to the context of our project. The text we wish to process is that of the news article body received our Trove requests. The main concepts we utilise when cleaning and preprocessing our data are those of tokenization, lemmatization, and stop word removal.

3.2.1 Cleaning

```

1  # Remove Markup
2  soup = BeautifulSoup(text, features="html.parser")
3
4  for script in soup(["script", "style"]):
5      script.extract()
6
7  text = soup.get_text()
8
9  # Remove reg. ex. / punctuation, this also removes the hyphen in
10 hyphenated words i.e freeze-dry -> freeze dry
    text = re.sub(r'[\W]', ' ', text)

```

Listing 3.4: Removing Tags

Before we begin tokenization, we first remove any hypertext markup language (HTML) tags and punctuation. As seen in the above code, we remove HTML tags by using BeautifulSoup, which is a Python library specialised for pulling data out of HTML files, usually used for scraping information from the internet [23]. We then remove punctuation such that only alphanumeric characters remain. From table 3.1 we can see the effects of this.

3.2.2 Tokenization

The next step is to turn our words into separate entities, or in other words, tokens. We use the ‘word_tokenize’ function from the NLTK library to segregate our text. In the code below we see the tokenization of our text, and also the replacement of any upper case letters with their lower case counterparts. This is so that there is no distinction between the same words that have capitalised letters, and words that do not have capitalised letters. The trade off for doing so is that we lose the ability to distinguish proper nouns from normal words, such as a person’s name or the name of a place. Fortunately, we are not currently interested in these proper nouns when detecting events.

The final result of the below code is a list of words which we are then able to remove stop words from and lemmatize, as seen in table 3.1.

```
1 # Tokenize and transform into lower case
2 text = word_tokenize(text)
3 text = [w.lower() for w in text]
```

Listing 3.5: Tokenization

3.2.3 Removal of Stop Words

Removal of stop words is an integral part of preprocessing text, as we are able to remove many redundant terms and thus lower overall dimensionality of the final dataset. We use the stopword list from NLTK as our list of words to remove [24]. We are able to see the result of this in table 3.1, where ‘here’, ‘s’, and ‘some’ are removed. In the case of ‘s’ being removed, although it does not belong to the stopword list, it is a token of character length 1, which is also a removal condition.

We implement what Baradad and Mugabushaka (2015) suggests, which is the incorporation of specific corpus stopwords to further optimize our algorithm efficiency [13]. Investigation of the popular terms further on in the project indicated that there were some popular nonsensical words due to the Trove OCR misidentifying letters. We also append these to the stopword list, such that these are also removed from our text. As seen below the most common words were terms such as ‘tho’ and ‘tbe’, which are assumed to be the incorrectly identified word ‘the’. From this stage, we move onto the normalization of text using either stemming or lemmatization.

```
1 # Remove stop words
2 stop_words = set(stopwords.words('english'))
3 newstopwords = ['tho', 'mr', 'tbe', '000']
4 stop_words.update(newstopwords)
5 filtered_text = [w for w in text if w.lower() not in stop_words and
w not in string.punctuation and len(w) > 1]
```

Listing 3.6: Removing Stop Words

3.2.4 Normalizing Text

As revision of surrounding literature suggests, we prefer the usage of lemmatization to that of stemming. This is because lemmatization produces actual dictionary words, which we will be using as features further on in the project. We also prefer lemmatization because we are interested in the words themselves, and not just the data that they create. From table 3.1 we are able to examine the major difference between the two methods in action. As evidenced by Liu et al., the term ‘generous’ is transformed into a nonsensical word ‘gener’ when stemmed, and is untouched when lemmatized [11]. Our code to used to lemmatize our tokenized text is as follows.

```
1  # Lemmatisation
2  lemmatizer = WordNetLemmatizer()
3  lemmatized_text = ' '.join(lemmatizer.lemmatize(token) for token in
4  filtered_text)
5  lemmatized_tokenized_text = word_tokenize(lemmatized_text)
6
7  return lemmatized_tokenized_text
```

Listing 3.7: Lemmatization

3.2.5 Calculating Frequency

The final values we need to calculate in order to complete our aforementioned dictionary for each article can now be calculated since we have processed the text. We again use an NLTK function called ‘FreqDist’, which is fed tokenized text and returns a frequency distribution of terms.

From this we are able to gather the last three values necessary to complete our dictionary. Firstly, the term count, which is simply how many tokens exist in the list that has been inputted. Secondly, term frequency which is the number of appearances of each unique token. Lastly, the term frequency divided by term count, we calculate this value because there is a high variance in article length. We see that term frequency values are higher in articles with higher term counts.

```
1 def compute_term_frequency(tokenized_text):
2     """Calculate term count, term frequency and term frequency versus
3     term count.
```

```
4      :param tokenized_text:
5      :return: return frequency of words and frequency of words against
6      amount of words
7      """
8      # Frequency distribution
9      term_count = len(tokenized_text)
10     i = 0
11     frequency_distribution = FreqDist(tokenized_text)
12     tf = []
13     while i < len(frequency_distribution):
14         # Where [i][1] is the frequency of the word
15         tf.append((frequency_distribution.most_common()[i][0],
16         frequency_distribution.most_common()[i][1] / term_count))
17         i = i + 1
18     return term_count, frequency_distribution, tf
```

Listing 3.8: Calculating Frequency

3.3 Storing Information

The last step necessary in order to complete this subgoal is the storage of the data. This is important as retrieval and preprocessing of large amounts of textual data often takes an extended amount of time. We utilise MongoDB, which is a document-oriented database program to locally store our data. Doing so is uncomplicated since we have already correctly organized and compiled our preprocessed historical articles.

Once both preprocessed lists from PerilAUS and Trove are stored, we will have successfully completed our first goal. With our final database constructed, we can begin feature selection and subsequently construction of our event detection models.

Storing PerilAUS Data

Below we outline the function used to read, format, and store the PerilAUS file. We utilise PyMongo, a MongoDB API that allows straightforward interaction with our local database. As seen in the below code, we specify our local database parameters to establish a connection, and use the provided inbuilt commands to insert and extract data. Using function `read_csv` as seen in example 3.3 to return the data in a list form, we do not need to change the format of our data to insert it, as MongoDB accepts list values as inputs.

```
1 def store_csv_data(filename):
2     """ Input filename, read and process csv and then store into MongoDB
3         database
4
5         :param filename:
6         :return:
7         """
8     client = pymongo.MongoClient("mongodb://localhost:27017/")
9     db = client['perilAUS']
10    col = db['events']
11    data = read_csv(filename)
12    col.insert_many(data)
```

Listing 3.9: Storing PerilAUS Data

Storing Preprocessed Trove Data

Next we store the preprocessed article text into a separate collection. The article storage function follows a similar format to that of the PerilAUS storage function. We are similarly storing a list, so there is no need to convert data formats.

```

1 def extract_and_insert(trove_key, search_terms, start_year, end_year,
2   database_name, collection_name):
3     """ Extract information from the Trove API, process it, and insert
4     it into the MongoDB database
5
6     :param collection_name: name of collection information is being
7     stored to
8     :param database_name: name of database information is being stored
9     to
10    :param end_year:
11    :param start_year:
12    :param search_terms:
13    :param trove_key
14
15    :return:
16    """
17    i = 0
18    client = pymongo.MongoClient("mongodb://localhost:27017/")
19    db = client[database_name]
20    col = db[collection_name]
21
22    skipped_years = []
23
24    while i <= (int(end_year) - int(start_year)):
25        year = int(start_year) + i
26
27        print('Collecting articles from year:', year)
28        art_col, skipped = new_process_search(trove_key, 'newspaper', '
29        Article&bulkHarvest=true', search_terms, year,
30                                           year, '*',
31                                           100)
32
33        print(len(art_col), 'items inserted into database:',
34              database_name, 'and into collection:', collection_name)
35
36        inserted_list = art_col
37        col.insert_many(inserted_list)
38
39        if skipped:
40            skipped_years.append(year)
41        i += 1
42
43    print('\t'.join(map(str, skipped_years)))
44    return

```

Listing 3.10: Storing Preprocessed Trove Data

Year Range	Unfiltered Articles	Articles after filter
1900-1909	31195	5735
1910-1919	44792	8083
1920-1929	31300	6792
1930-1939	48370	10062
1940-1949	30186	5584

Table 3.2: Articles before and after filtering

We input the desired range of years we wish to preprocess and store results from, and the local database and collection we will store them in. We use print statements to update the user in the case of encountered issues or successful storage. As clarified in 3.1, there are some years that have inaccessible data due to the 'nextStart' error. In the above function, we take note of these years that this occurs, so that we will be able to refer to these in the case we experience skewed results further on in the project. The final result we expect from running this function is preprocessed articles from the specified years stored in a single specified collection.

3.4 Filtering Function

The thesis supervisor had provided an article ranker function that further filters the articles gathered from Trove, and sorts the articles into various ranks according to how relevant they are to the disaster itself. The article ranker is a set of machine learning models that estimates the importance of an article relative to the occurrence of a natural disaster recorded in the PerilAUS database. A manual set of training/testing data was created by going through each trove article published within a few weeks of a past cyclone occurrence and ranking it from 0 to 5 stars, depending on the amount of information regarding fatality and damage that it provides.

It is a three step process composed of feature engineering, a filter model, and a ranker model. The feature engineering uses word embedding technology to transform texts into feature vectors. The feature vectors are built within the context of the cyclone occurrence ensuring local word semantics. These feature vectors are then fed into the random forest model trained to filter out articles unrelated to the cyclone occurrence. Finally, the filtered articles are star-ranked using a second random forest model trained on the manually constructed dataset. When generalised, this process provides a very efficient method to filter and rank historical text to discover detailed information regarding past natural disasters.

The results of the filtering are significant, as when we remove 0 star articles which are redundant or otherwise irrelevant articles from the dataset, we are then left with a much

smaller pool. This can be evidenced in the table `reftab:filter-table`, where we are left with approximately 20 - 30% of our original articles. Removal of this redundant data removes much ambiguity from our final results, whilst also further enhancing the importance of specific features we will be implementing in the following chapter of feature selection.

Chapter 4

Feature Selection

This section details our feature selection process throughout the project. We will utilise the already gathered data from our efforts in chapter 3, extracting data from our locally stored database. Feature selection is deemed to be the most complicated goal to complete, as it is the procedure of defining the inputs that construct our event detection models.

Feature selection is the process of selecting independent variables that will contribute to our final prediction variable, there are multiple techniques used to choose these variables. We use a combination of univariate selection and feature importance to decide what features will be input into the final model. This chapter is split into multiple sections, the first of which outlines the main method we use to solve the selection problem. The next two sections describe the major feature types we test and input into our models; article frequency, and word frequency. A basic illustration of the experimentation loop is seen in fig. 4.1.

4.1 The Sliding Window Method

We look to Dietterich (2002) who outlines four primary strategies for solving the feature selection problem [19]. We aim to emulate a mixture of the sliding window strategy, and the strategy which involves the removal of low-scoring features. The sliding window method is essentially specifying a length of a data range that “slides” across our data, capturing each window’s information. Our data in this case is a large list of dates, each containing information that has been pre-processed as discussed in chapter 3. We use these snippets of information to calculate averages, and create a new datapoint, which will be used to extract our features.

The reason we implement this strategy is because our initial investigations yielded

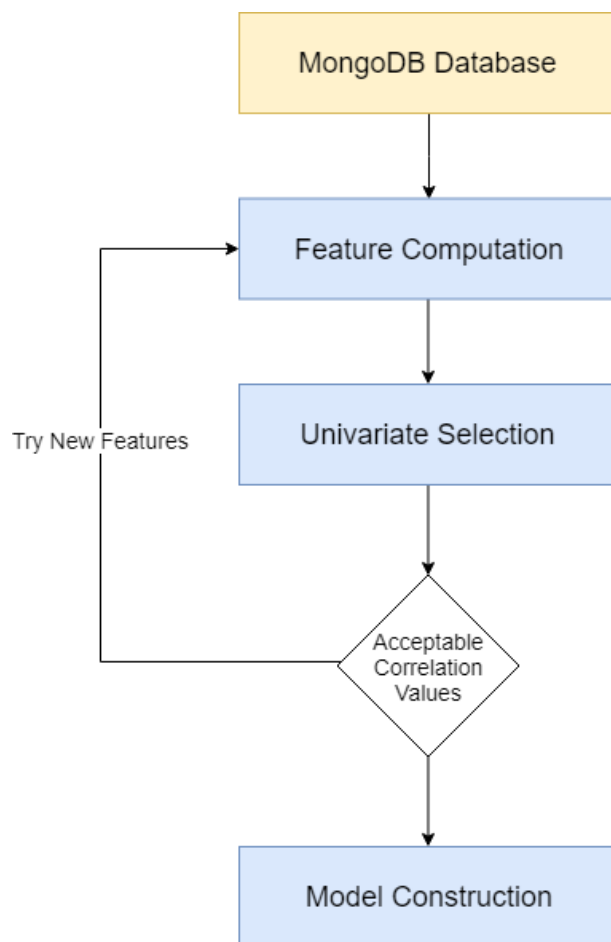


Figure 4.1: Text Preprocessing Flowchart

Date	Article Frequency	Event Occurrence
01/01/1900	13	0
02/01/1900	14	1
03/01/1900	37	0
04/01/1900	45	0

Table 4.1: Example Dataset

that a single day's worth of information as a data point had very little information that correlated with event occurrences. This can be attributed to the fact that the articles published for a natural catastrophe would not be concentrated over a single day, but rather over a span of dates subsequent to the catastrophe. We evidence this claim by correlating the frequency of articles published against event occurrence using the Pearson standard correlation coefficient which comes with the pandas library [25]. We demonstrate what our dataset would look like in table 4.1.

Our event occurrence values are set as either a 0 or 1, with the former indicating a normal day, and the latter indicating an event occurrence. The correlation value returned is always a value between -1 and 1. -1 indicating that there is a linear negative relationship between the values, and 1 meaning there is a linear positive relationship between the values.

This correlation test is conducted from on dates spanning from 1900 - 1903. As expected, a range of a single day of article frequencies correlated against event occurrences results in a very poor result of approximately 0.037, indicating there is little to no correlation between the two values. We test subsequent ranges up to 17, alongside a weighting system that proportionally decreases the value of following days in relation to proximity from the beginning date. This resulted in the figure 4.2, in which we see that we experience rapid growth from 0 to 4, then the value momentarily plateaus and begins to decrease after 14.

Although our highest value is experienced at a range of 14, we use the range of 4 for all of our subsequent models. This is because further investigation had yielded that models with data points at a range of 14 were proficient in predicting events with high impact, but often missed those with low impact. Models based on data within a range of 4 were superior in detecting the presence of events with little impact, whilst also maintaining the high prediction rate of high impact events. This makes logical sense as events with small impacts would result in very few days of subsequent articles, whilst the opposite would be true for large events.

From this investigation we have successfully proved a slight, but significant positive correlation between article frequencies and event occurrences, created a successful weighting schema, and also have arrived at our optimal sliding window range of 4.

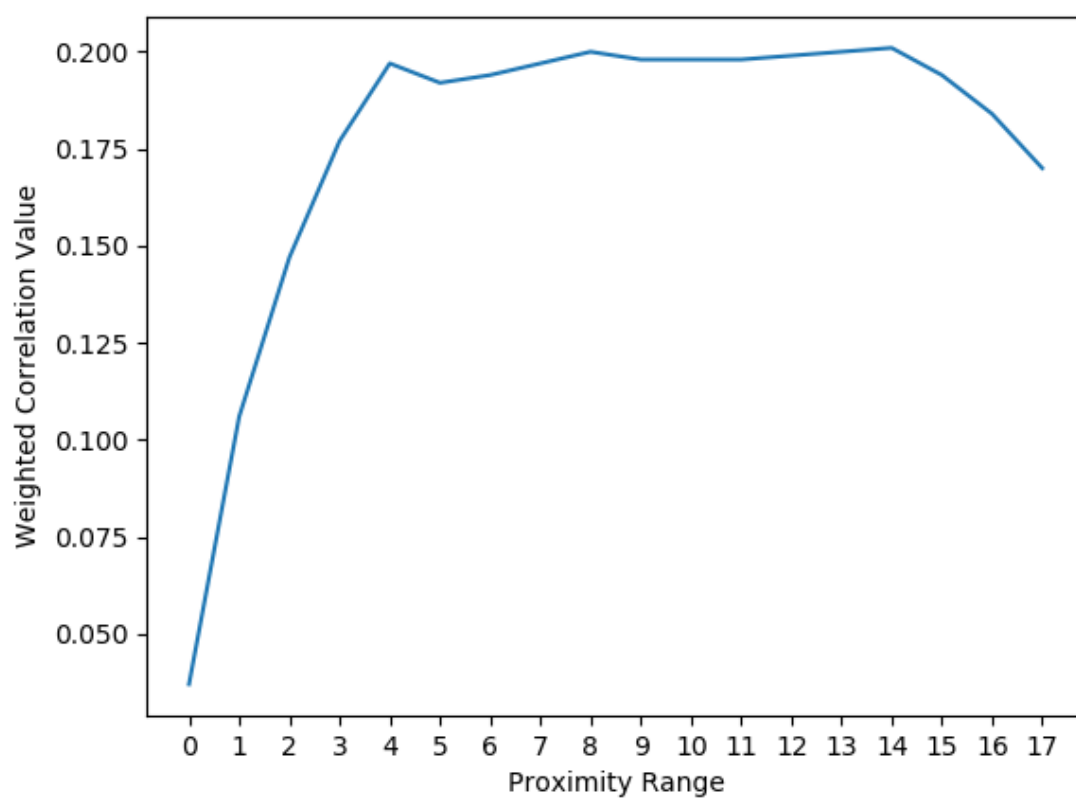


Figure 4.2: Proximity Range versus Weighted Correlation Values

4.2 Article Frequency as a Feature

Article frequency has been identified as one of the major feature categories we implement when constructing our dataset which will be the basis of our models. The basis of this claim is that there should be more overall articles in the subsequent days after an event occurrence, and that thus there should be some correlation between the two variables. We derive multiple features from this category, which are to be explained and demonstrated throughout this section. These features are; unfiltered article frequency, filtered article frequency, frequency of each ranked article, and the average of the ranked articles. These are judged by the aforementioned methods of univariate selection and feature importance.

The first point of contention is that of unfiltered article frequency and filtered article frequency. There is expected to be much more redundant data from the unfiltered article category, and thus more overall noise. This then leads to an overall lower correlation value with event occurrence, and is thus less valuable than our filtered article category. This is evidenced through examination of the corresponding correlation values. We experience a marginal, but important increase in correlation value when comparing the two features. In which unfiltered article frequency returns a result of -0.19, filtered article frequency returns -0.23. It can be seen that the unfiltered article frequency feature has an overall lower correlation with event occurrence, this not only provides further evidence that the filtering function succeeds in removing redundant data, but also succeeds in improving our results.

Each ranking category from 1 - 5 is then input as a feature. The expected result is that there will be an increase in correlation values proportionate to the ranking category. This is because higher ranked categories are expected to have information that is more relevant to the natural catastrophe. We verify this assumption by plotting the frequency of articles within the aforementioned ranking categories and calculating their associated correlation values. For the sake of clarity and to avoid clutter, we only plot one star and five star frequency against event occurrence. This model uses our amended event occurrence values which are to be outlined within 5. We see in fig. 4.3, contrary to what logic would dictate, that there is a very miniscule difference in the relation between one and five star article frequencies to event occurrence. The only noticeable difference is that the spike in each frequency is different for some events. Our correlation values also prove the same as seen in table 4.2, where we see a small decrease in value rather than an increase proportionate to rank. This could be attributed to multiple causes, the most likely being that there is not a large difference between rankings, or that events are likely to have more articles published regardless of ranking.

Our final feature derived from this category is an amalgamation of the aforementioned features, in which we assign a point-based system to each ranking, and take the average of amount of points scored within the specified range. The point system is simple, in which the articles are worth their respective ranking in points. For example a one star article

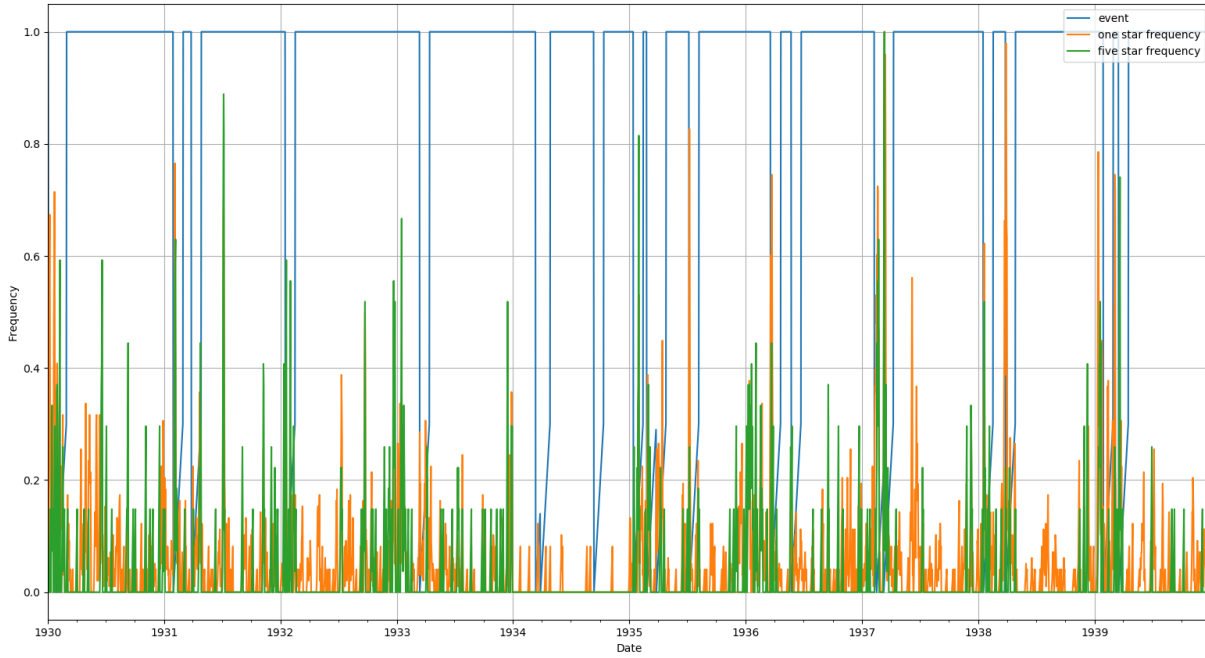


Figure 4.3: One and Five Star Frequencies Against Event Occurrence 1930 - 1939

Ranking	Correlation Value
One	-0.32
Two	-0.28
Three	-0.26
Four	-0.20
Five	-0.20

Table 4.2: Correlation Values of Article Frequencies of Rankings

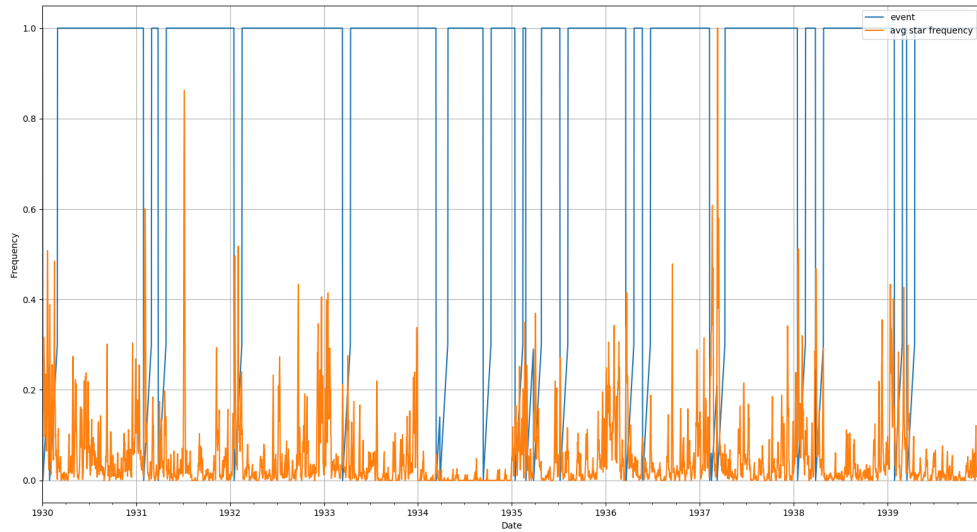


Figure 4.4: Average Star Ranking Against Event Occurrence

is worth 1, whilst a five star article is worth 5. This is graphed against event occurrence in fig. 4.4, with a resulting correlation value of -0.29. This value is similar to that of the other star rankings, but is a worthwhile feature to include.

4.3 Word Frequency as a Feature

Word frequency allows a deeper investigation into the correlation of specific terms, and the frequency of their usages, with event occurrence. The logic behind this is that some terms will tend to appear more frequently after a natural catastrophe occurs, and that by identifying these spikes in frequency, we can identify the events themselves. The features we derive from this investigation are all of the same format, yet each contains the information of a different term. In this section, we outline the process in which we arrive at our final feature type, and how we choose what words will be input as features. For the sake of brevity, our examples use the same word “cyclone” as our input term.

4.3.1 Determining the Input Type

Our rudimentary analysis of articles allows our application to count the frequency of specified terms over a particular range. For example, we can choose a certain date, and

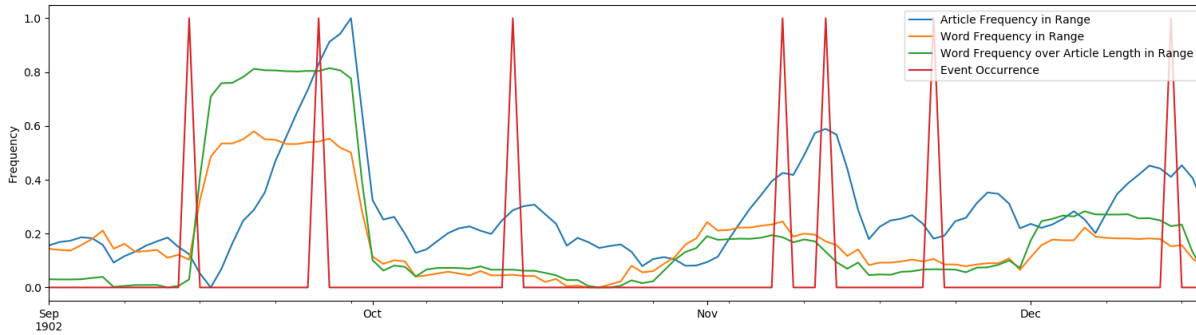


Figure 4.5: Frequency Types against Event Occurrence September 1902 - December 1902

count the amount of times the word “cyclone” appears in articles published over the next few days. We designate this input type as the pure word count. The biggest issue we encounter with this is that this method does not take into account the highly volatile amount of words a single article can have. The average word length of an article from 1930 to 1939 is 78.45 words, whilst the longest article found in that range is 1323 words. The issue we experience during analysis is that our function can mistake large articles, with a high occurrence of our chosen terms, as articles indicative of an event occurrence. We can counter this issue by taking the article word count into the equation when computing our features.

The above solution yields a more informative result, evidenced by the snippet of a graph from 1902 where a large event occurs as seen in fig. 4.5. We see that the first event in September had a high impact, proven by the sharp spike in each variable. We see that dividing word frequency by article length yields a higher result after the event, meaning that there was a very high concentration of the term ‘cyclone’ occurring in this period. Circumstances such as this are rare as seen in the other events during this time period, meaning that our new solution is still lackluster in predicting the occurrence of an event.

We look towards term frequency-inverse document frequency (tf-idf) values as our new feature input. We use sklearn TfidfVectorizer to compute these values [26]. These values are used to determine the relevance of terms within a specified corpus of information. The term value increases proportionally to the amount of times the term appears and decreases proportional to the amount of documents that contain said term. We can use this to identify the relative importance of specific terms, and their prevalence within our specified 4 day sliding window range. We take the average tf-idf value of our specified term within our data range, and input this as our feature. We find that our correlation value for this feature is much lower than both pure term frequency, and term frequency divided by article length as seen in table 4.3. It is not until we begin comparing features by feature importance values returned by our machine learning models in chapter 5 that we find that tf-idf values are the most helpful out of our three options when it comes to

Input Type	Correlation Value
Pure Word Count	-0.26
Word Count vs Article Length	-0.28
Tf-idf Values	-0.04

Table 4.3: Correlation Values of the term ‘Cyclone’ Various Word Frequency Input Types

predicting event occurrence values.

4.3.2 Determining the Important Terms

Now that we have devised our three word frequency input types, we must determine what terms we input into the model. There are multiple methods to decide which words are the most indicative of an event, each with different advantages and disadvantages. The first method we discuss is the inclusion of ALL available terms, followed by the removal of low scoring terms. This is the most time consuming and most exhaustive method, as there are hundreds of unique terms even after the preprocessing of each article. This would be unfeasible for this project due to the lack of necessary resources, so instead we look to a faster, but less inclusive method.

Using our tf-idf computation function, we can find the most important terms within any given data range. We run this function 4 days after every listed event occurrence, and take the average of each term to find the terms with the highest scoring average. For example, the top scoring tf-idf terms for 1930 - 1939 are as follows in table 4.4. We notice that similar terms appear in other year ranges, and use these as our feature input terms. The advantage of this method is that it is much less resource intensive, and that we are directly investigating the terms that are most concentrated after each event occurrence. The average tf-idf values of these terms will be used as features to create the machine learning models.

Ranking	Term
1	cyclone
2	weather
3	last
4	wind
5	town
6	rain
7	night
8	yesterday
9	damage
10	storm

Table 4.4: Highest scoring tf-idf terms within 4 days of event occurrences 1930 - 1939

Chapter 5

Data Modelling and Evaluation

Now that features have been gathered, we are able to construct our machine learning models. These models will take the features as independent variables, to predict the dependent variable, the event occurrence. In this chapter, we describe the methods in which we construct, improve, and evaluate our datasets and models. This chapter is split into two sections, the first of which outlining the different model types we experiment with, and the second to evaluate the effectiveness of these models in achieving our main goals. The flowchart for this chapter can be seen in fig. 5.1, with the same experimentation loop as seen in previous chapters such that only the most indicative features remain.

5.1 Dataset and Model Construction

Our dataset as mentioned briefly in chapter 4 consisted of various features which exist as independent variables, and one dependent variable, the event occurrence. These variables exist for every given date within the collected dataset itself, in that every date has corresponding variables. These variables are collected from the next four days after the date itself, as per the sliding window method referenced in section 4.1. Recall that the event occurrence variable was designated as always either a 0, or a 1. In which the former meant that the day had no event occurrence, and the latter indicating an event had occurred.

The above dataset when modelled would use the inputted features to predict event occurrence. The issue with this is that it is a classification problem for a heavily unbalanced dataset. This is because the dataset consists of an extremely high amount of non-event occurrence dates, and thus an extremely low amount of actual event occurrences.

According to the listed perilAUS events for tropical cyclones as described in table 5.1, we show the average number of natural catastrophe occurrences within a ten year span

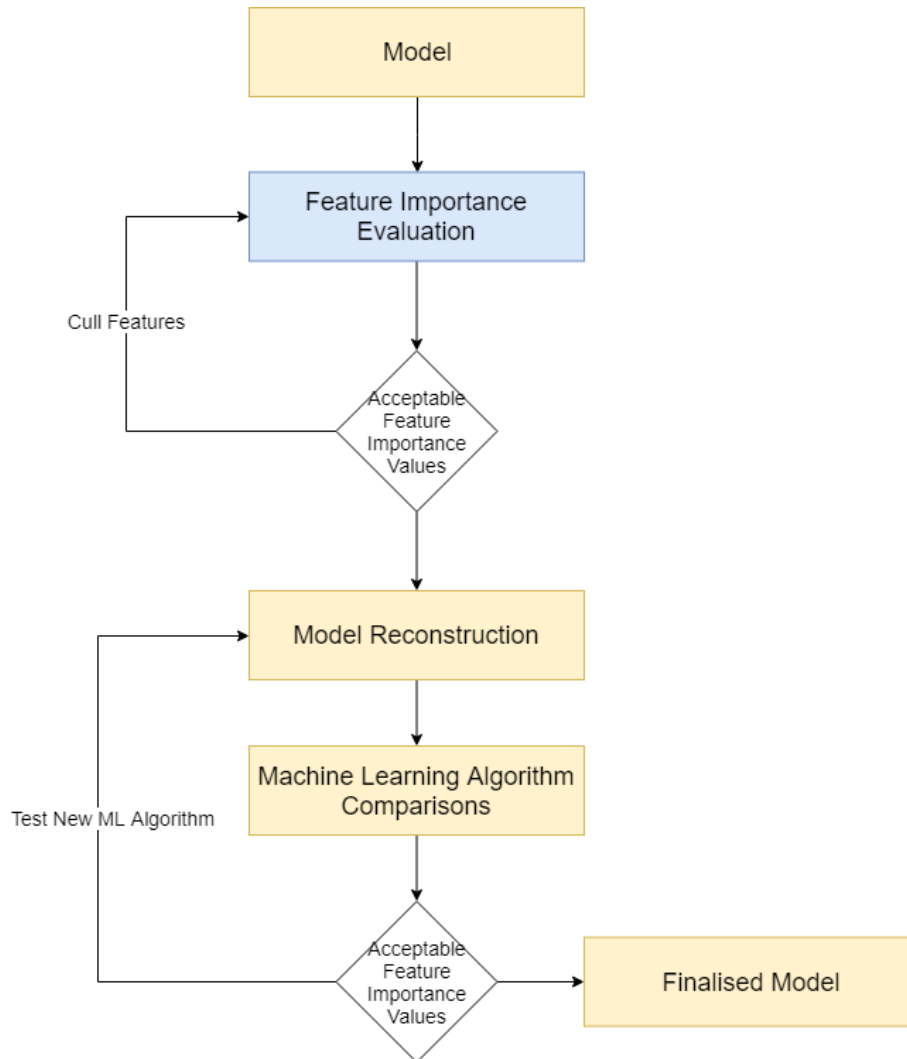


Figure 5.1: Model Construction and Evaluation Flowchart

Year Range	perilAUS Number of Listed Tropical Cyclone Events
1900 - 1909	16
1910 - 1919	35
1920 - 1929	26
1930 - 1939	25
1940 - 1949	20
Average	24.4

Table 5.1: perilAUS Number of listed tropical cyclone events every ten years 1900 - 1949

of a very low 24.6. This means that within an average of 3650 days, only an approximate 0.007% of days have a listed event occurrence. The issue with this is when constructing a classification model, the model will heavily lean towards predicting every date as a non-event occurrence, regardless of the input features, as to achieve a very high accuracy score. This is evidenced by our very early random forest classification models achieving an accuracy score of 98.6%. This type of model does not achieve our goal, as although it is correctly predicting non-event occurrences, it does not predict any real event occurrences correctly.

We attempted to combat this issue with a mixture of methods. One of the methods we strived to implement was to under-sample our dataset. Under-sampling is the process of deleting instances of the over-represented class, which in our case is the non-event occurrence dates. The second technique we employed was to change our overall problem from classification to regression. We do this by changing the rules of our event occurrence variables, from a simple binary system to an incremental system. When an event occurs, the date's event occurrence value is denoted with a 0, with each subsequent days event value increasing by 1 for each day further away from the origin point, to a maximum of 30. Dates outside of this range are all removed as per the under-sampling method, or have their event values denoted with 100. From this, we create two separate datasets. The first a mixture of undersampling and the change to the event values system as shown in fig. 5.2 The second implements only the change to the event value system as shown in fig. 5.3. We investigate the effectiveness of both of these datasets, and decide which is more suited to achieve our goals in section 5.2 by looking at the predicted values of the models trained on each dataset.

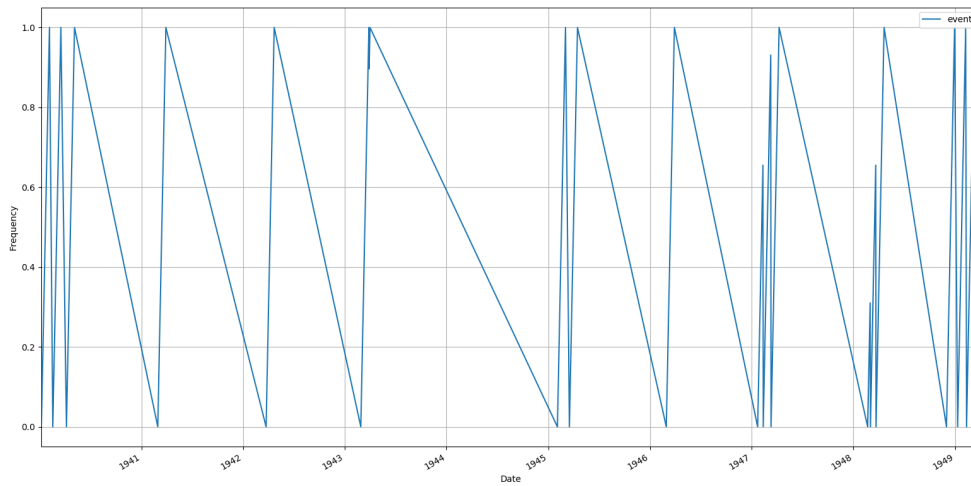


Figure 5.2: Regression Dataset with Undersampling 1940 - 1949

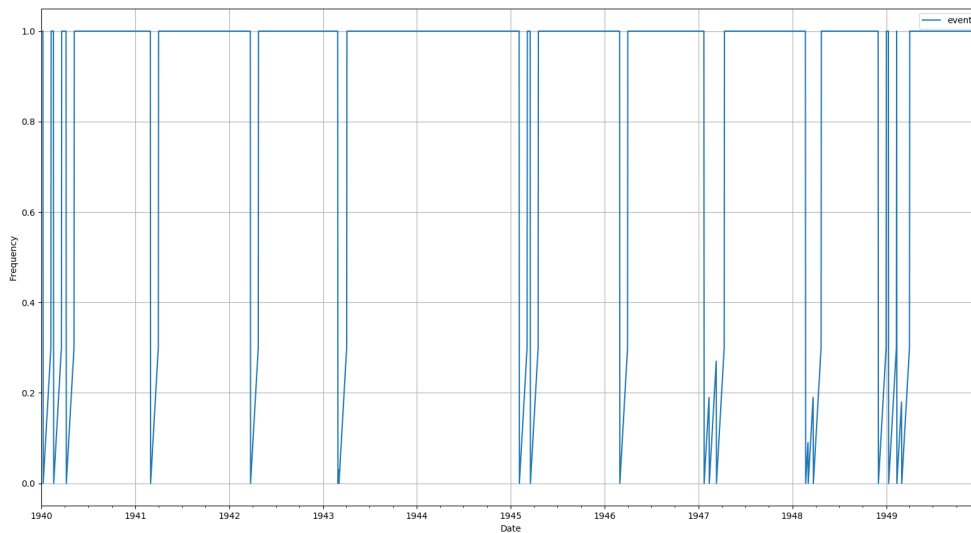


Figure 5.3: Regression Dataset without Undersampling 1940 - 1949

5.2 Model Evaluation

In this section we train models based on our improved datasets, and evaluate the validity of each model. From our evaluations we seek to further refine the model, which we attempt to do by experimenting and changing different aspects of our models. These aspects include the removal and addition of features using the returned feature importance values from our models, and changing of the machine learning algorithm used to build the model itself.

As discussed previously, a high accuracy score does not necessarily equate to a model that achieves our goals. In order to reliably evaluate our models, we look towards other performance metrics. We use precision, recall, and F1 scores to determine whether our models are achieving our intended goals.

Precision is the ratio between correctly predicted positive results and the total amount of predicted positive results. In this case, our correctly predicted positive observations would be correctly identified real event occurrences. Therefore, the lower false positives our model returns, the higher the precision score of our model is.

$$Precision = \frac{True\ Positives}{True\ Positives + False\ Positives} \quad (5.1)$$

Recall is the ratio between our true positives predictions and the total amount of actual positive cases, which is the sum of true positives and false negatives. False negatives in this case are the incorrectly predicted non-event occurrences, in other words, dates that are actually event occurrences but have been classified as non-events.

$$Recall = \frac{True\ Positives}{True\ Positives + False\ Negatives} \quad (5.2)$$

Lastly, the F1 score is a function of both precision and recall. We use these measures as our performance metrics because our dataset is heavily unbalanced, and because we do not hold much interest in the amount of true negatives returned.

$$F1\ Score = 2 * \frac{Precision * Recall}{Precision + Recall} \quad (5.3)$$

We classify a date as an event occurrence if the predicted score drops below a certain specified classification threshold. Depending on how low the score is relative to this threshold, the higher the confidence interval of that particular event is. For example in fig. 5.5 we specify the threshold as 0.4, predictions closer to 0 would have a higher

confidence interval. The performance metrics for this figure is as seen on table 5.3, and are quite low. Throughout this section we describe how these values are improved.

It is imperative to note that false positives in this scenario may only be false positives in relation to the perilAUS listed events database. This means that there may be missing events yet to be listed, that are correctly predicted by our machine learning models. These would technically be true positives.

5.2.1 Comparing Models Trained on Undersampled Data versus All Data

In order to compare the differences between our two datasets we created in 5.1, we train two separate models using the sklearn random forest regressor algorithm [27]. In this instance we build our models from ten years worth of data spanning from 1910 to 1919. The data is then split into an 80/20 ratio, the former for training the model, the latter for testing and fine tuning the model. We then further test these models on another decade of data, and compare the results.

Fig. 5.4 depicts the prediction values of our model trained on the undersampled data, from this we already see that we cannot draw any discernible event predictions. This can be attributed to the fact that we are testing a model trained on data gathered only within a certain range of event occurrences, in this case 30 days. Thus once it is tested on a full set of data, we see that it is not trained to handle the abundance of non-event dates and provides non-informative predictions. This type of dataset is unsuitable to achieve our overall goals. Fig. 5.5 provides much clearer results. We notice quite a few overlapping predictions with listed events. The difference is attributed to the model without undersampling handling the massive amount of non-event occurrence dates better due to being already trained for such a scenario. We decide to use the model trained on all data as our foundation moving forward, and now look towards tweaking the features being input into the model.

5.2.2 Altering Features based on Feature Importance

In chapter 4 we consider two methods to assist in selecting relevant features, univariate selection and feature importance. We are now at the stage where our models return feature importance, and we can use this information to cull unnecessary features or modify existing ones. We are able to determine this feature importance by simply calling an already built in sklearn function. The model shown in fig. 5.5 returns the following feature importance list as seen in table 5.2. We see that the average star frequency feature is seen as almost twice as important as the next best feature, and that the other

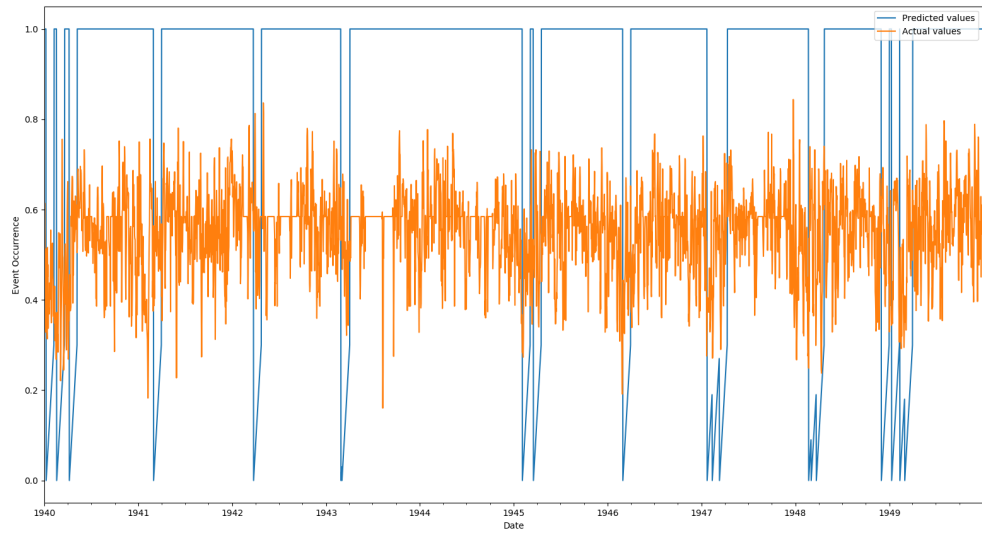


Figure 5.4: RF Regression Model Predictions Trained on Undersampled Data 1910 - 1919, Tested on Data 1940 - 1949

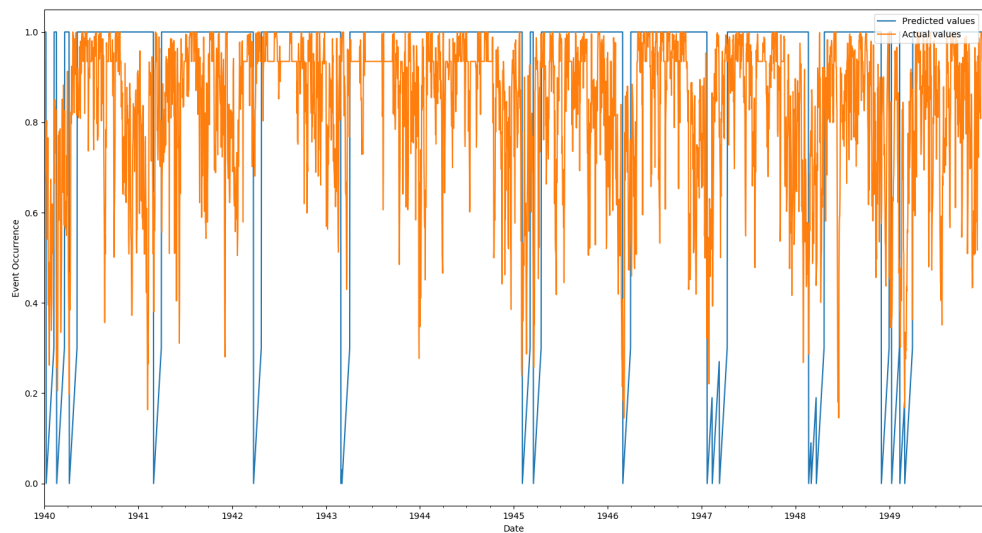


Figure 5.5: RF Regression Model Predictions Trained on All Data 1910 - 1919, Tested on Data 1940 - 1949

star and article frequency features have little to no bearing on the final result.

We are able to cull these features in our next model iteration, providing new predictions and performance measures. We call this model the non-article frequency model, and now compare the predictions as seen between fig. 5.5 and fig. 5.6. Whilst keeping our classification threshold the same value at 0.4, we see an increase in all three performance metrics. We attribute this change to the reduction in overall noise of the model, thus increasing model effectiveness.

We continue to cull features until we are left with a single feature, the average star frequency. This model is dubbed the average star model, and from 5.7 we observe more pronounced predictions overlapping with event occurrences. Table 5.3 indicates a higher recall rate than the previous two iterations, but a lower overall precision score. This is due to the classification threshold being high enough to detect a higher ratio of false positives compared to true positives. We are able to lower this threshold due to the pronounced predictions produced from this model to 0.2, but results indicate an ever lower precision score and consequently a lower f1 score. This is because the proportion of true positives detected has decreased more than the proportion of false positives.

When choosing from these models, we note that the non-article frequency model has the highest f1 score, and is in every way an improvement over the basic model which includes all features. We find that the average star frequency model with a low classification threshold of 0.2 performance is too poor compared to the same model with a higher threshold of 0.4. We choose to use the average star frequency model with a threshold of 0.4 as our final model, even though it has a lower f1 score than the non-article frequency model. This is because in order to meet the previously set requirements, we prioritize gaining a higher recall score over a higher precision score. This ensures that the model has correctly predicted the existence of already listed perilAUS events, and still has the potential to detect previously unlisted events. It is unfeasible to investigate all of the false positives to determine if they are actual unlisted events, but rudimentary evaluation shows promising results. All of the currently manually investigated articles describe a previously unlisted cyclone event, but most describe the cyclone as either passing by or non-damaging. There are a few events that are unlisted that newspaper articles describe have done damage, and these are the most promising indicators of tangible results.

Feature	Importance
avg star frequency	0.144630
"wind" tfidf value	0.073262
"cyclone" tfidf value	0.059883
"damage" tfidf value	0.059565
"heavy" frequency	0.058097
"storm" tfidf value	0.056111
"last" tfidf value	0.055948
"wind" frequency	0.055889
"rain" tfidf value	0.050268
"heavy" tfidf value	0.050060
"yesterday" tfidf value	0.049229
"night" tfidf value	0.048605
one star frequency	0.047913
"town" tfidf value	0.044177
three star frequency	0.038793
article frequency	0.035348
two star frequency	0.028100
five star frequency	0.025540
four star frequency	0.018583

Table 5.2: Feature Importance returned by model trained on data from 1910 - 1919 and tested on data from 1940 - 1949

	All Feature Model	Non-Article Frequency Model
Precision	0.167	0.2
Recall	0.35	0.4
F1 Score	0.226	0.286
Classification Threshold	0.4	0.4

	Average Star Frequency Model	Average Star Frequency Model
Precision	0.144	0.137
Recall	0.75	0.5
F1 Score	0.242	0.215
Classification Threshold	0.4	0.2

Table 5.3: Performance Metrics of RF Regression Models Trained on All Data 1910 - 1919 and Tested on Data 1940 - 1949

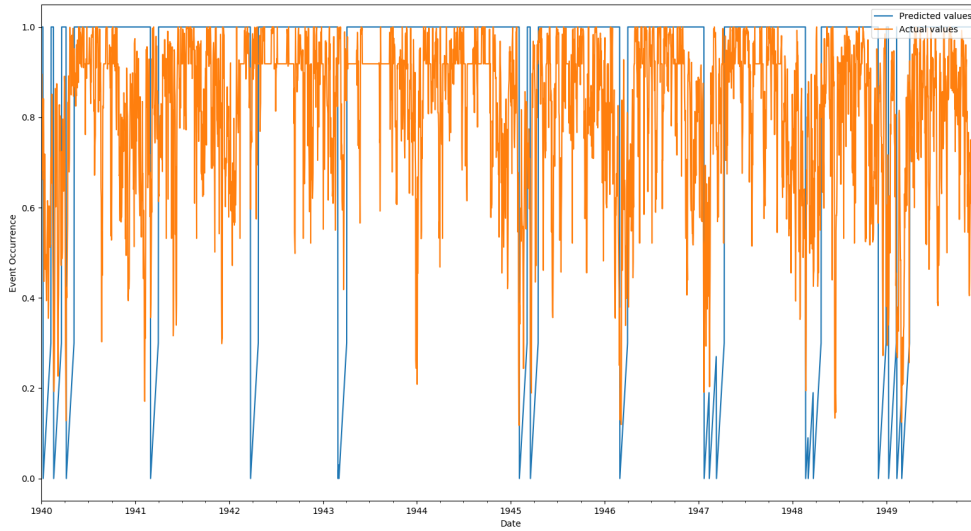


Figure 5.6: RF Regression Non-Article Frequency Model Predictions Trained on All Data 1910 - 1919, Tested on Data 1940 - 1949

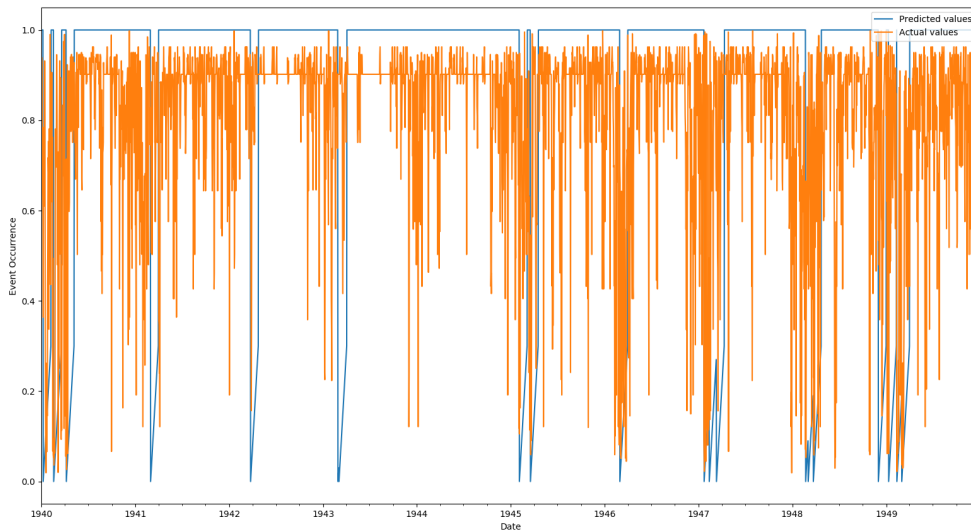


Figure 5.7: RF Regression Average Star Frequency Model Predictions Trained on All Data 1910 - 1919, Tested on Data 1940 - 1949

	RF Model	SVM Model	KNN Model	KNN Model (L)
Precision	0.144	0.147	0.118	0.157
Recall	0.75	0.5	0.8	0.65
F1 Score	0.242	0.227	0.205	0.252
Classification Threshold	0.4	0.4	0.4	0.1

Table 5.4: Performance Metrics of Different ML Regression Models Trained on All Data 1910 - 1919 and Tested on Data 1940 - 1949

5.2.3 Choosing the Machine Learning Algorithm

Background research represented different opinions on what machine learning algorithm would be best suited for our model. We experiment with three different types, RF, KNN, and SVM and decide which to use based on the returned performance metrics. The finalized average star frequency dataset is used as the foundation for the results found in table 5.4.

We see that the RF and KNN models contend closely with one another in respect to recall scores, whilst the SVM model falls behind. There is a large difference for the performance metrics of the KNN model when shifting the classification threshold down, this is because doing so removes many false positives and a minimal amount of true positives. The RF model is still more appealing in this scenario, as our project goal prioritizes the higher recall value since we are less concerned if there is a small increase in false positives in order for there to be a large increase in true positives. The RF model with the average star frequency dataset is chosen as the final model and product of this research project.

Chapter 6

Maintenance and Testing

It is essential to recognize the importance of the roles maintenance and testing have in the organization, efficiency, and overall completion of the research project. The project aims to demonstrate various techniques to keep to high standards.

6.1 Documentation

Documentation is key in maintaining both software code and research project readability. Functions can be difficult to comprehend without any provided description or usage hints, especially with the presence of duck typing in Python. Our code eliminates any confusion by attaching a docstring to every function that adheres to the reStructuredText Docstring Format, which Python proposes to be the standard markup format [28]. The format itself presents the:

- Function description
- Input arguments
- Return parameters

Another way to maintain maintainability during documentation is to modularisation of code. This is the process of organizing code into smaller chunks based on the task it performs. This method of coding allows for uncomplicated debugging, as it is much easier to pinpoint where the issue occurs. This method of coding allows the programmer to reuse functions for various different contexts, removing the need to rewrite code. Modularisation also assists in improving the readability of the code, allowing straightforward organization

and referencing, which is especially helpful in case researchers in the future wish to peruse through the code. This research project succeeds in heavily modularising functions, and reuses them when necessary to achieve organized and readable documentation.

6.2 Version Control

Usage of version control software provides many advantages. Manually storing changed versions of code and documentation is both tedious and prone to human error. Utilising version control software streamlines this process by automatically documenting the changes between versions. Version control software also organizes the previous changed versions and allows simple switching, the software also serves as a backup in case files are lost. This project uses both BitBucket and Github, which also allows the supervisor to easily monitor progress and thus maintain a steady form of communication.

6.3 Testing

This project aims to implement a variety of techniques to ensure that each aspect of the application functions as intended, but we accept that there are restrictions in some cases. Some aspects which the application tests are; database completeness, correct preprocessing, and model validation.

Extracting data from the Trove API is an arduous task as discussed in chapter 3, and is prone to many errors. These errors can cause repetitions of information to be stored, or for gaps to exist within the database itself. We attempt to remedy this by constructing our functions with these issues in mind, for example, the data storage functions take note of large voids, or missing articles when running, and reports this back to the user once the function completes. Some of these gaps in data are unable to be fixed as the issue is server side, these in particular are factored in when testing and constructing our models. Before preprocessing en masse begins, we ensure the functions output the expected result by running a few sample text examples and inspecting the results.

Our models rely heavily on testing to ensure they are outputting the intended results. This is evident in the previously mentioned 80/20 split of training and validation data, then further testing on completely new data to ensure we overfitting and underfitting is not occurring. For example, our models trained on data from 1910 to 1919 are tested on four more decades of data to ensure that results are consistent and not erroneous.

Chapter 7

Discussion

It is important to reflect on the overall results gained from the research project, and the methods implemented to attain these results. This chapter discusses difficulties and anomalies encountered at various stages of the project, the techniques used to remedy or avoid these issues, and how these affect the finished product. The issues encountered during data retrieval chapter were seen as the most difficult to deal with.

Data retrieval in this research project depends on a third party supplying information to the application to be manipulated such that tangible results can be extrapolated. The issue is that if the third party supplies erroneous data, or refuses to supply any data at all, there is little the application can do locally to avoid or mend this. Data was erroneous in that returned text from articles was often either incomplete or incorrectly spelt, this text may have been vital in the feature gathering process to predict events. As it is unfeasible to manually investigate each article text to correct spelling errors, we inevitably lose this information during preprocessing. If time and resources had allowed it, the application could have integrated an auto-correct function specifically to salvage nonsensical words, although this comes with a whole host of other issues. The Trove API often times refused to send data from specific timeframes due to unknown reasons, for example, the MongoDB database is missing all articles from 1907 - 1908. The only way this problem could have been avoided is if the application had another third party news aggregate supply information.

The feature selection process suffered from similar issues, in which there was too much redundant data being provided that results were beginning to be adversely affected. In the case of univariate feature selection, we found that, in the beginning there was very little correlation between article frequency and event occurrence. This correlation experienced a sharp increase when the article ranking and filtering functions were implemented and removed approximately 75% of all total articles from the database. Investigating feature importance as returned by our machine learning models, we found that a majority of

features being input to the model were unnecessary. Again, removal of these redundant features showcased a leap in performance metrics. The main takeaway being that more data only makes for a better model if it is useful, and that more features does not equate to a higher quality model.

Results returned from the final model indicate varying degrees of success. The model does indeed complete the objectives set out at the beginning of the research project, but does so with less precision than desired. For example, the final model tested on 1940 to 1949 returns an average of 16 of 20 correctly predicted listed event occurrences. The issue is that the model does so with a relatively high amount of false positives, averaging at 95. Of course, these are only false positives assuming the perilAUS database has accurately listed every event within this period of ten years. As described briefly during 5, manual investigation of the false positive predictions with high confidence intervals show very promising results. An example of this is the discovery of a previously unlisted cyclone event occurring on 1940-01-19, which is described as dealing an estimated 30,000£ to 40,000£ in damage costs [29].

We find that although the model has a much lower precision score than we would like, it is a vast improvement in efficiency over potential manual investigation of every date that is needed to verify the validity of the perilAUS database. The final output provides a much smaller area for investigation, averaging 95 days of 3650, which is a 97.4% decrease in search area. This can be further improved upon by taking into account the returned confidence intervals of the predictions. Taking all predictions with over a 90% confidence rating reduce the number of false positives to investigate to 11, as seen in table 7.1 which depicts the range in which the event may have occurred, the corresponding confidence intervals and article IDs. Investigating predictions with high confidence intervals lowers the overall manual investigation area to a miniscule 0.3% of 3650.

There were many adversities to face in order to complete the application, these are important to document in the case future researchers run into the same or similar issues. We recognize the limitations associated with depending on a single information source, and the methods in which we can improve our models performance metrics.

Begin Date	End Date	Confidence Interval	Article IDs
1940-01-16	1940-01-19	0.968150	40860681, 11272077, 169638252, 62423759
1940-01-20	1940-01-23	0.919047	98246715, 248232881, 162227877, 8196508
1940-03-05	1940-03-08	0.954775	247482775, 48343137, 46360684, 19276810
1946-02-23	1946-02-26	0.915375	212559175
1946-03-07	1946-03-10	0.919047	145014878, 62887273, 62887273, 98201635
1946-03-23	1946-03-26	0.917350	245394351, 245394351, 48466993, 1949266
1947-01-25	1947-01-28	0.919047	18011118, 18011118, 229999617, 17096225
1948-06-14	1948-06-17	0.925750	129903670, 158143779, 195094083, 195094
1949-02-10	1949-02-13	0.970100	205363912, 194897583, 52664808, 1809666
1949-02-14	1949-02-17	0.914900	18101082, 49928675, 224890197, 24772327
1949-03-03	1949-03-06	0.925750	117096590, 247719963, 49925404, 2649730
1949-03-07	1949-03-10	0.921675	194571438, 171106322, 63444549, 5689669

Table 7.1: Predicted Events with High Confidence Intervals and Corresponding Article IDs

Chapter 8

Conclusions

The goal of this project was to produce an application that accepts historical data as an input, and outputs the dates in which natural catastrophe events are most likely to have occurred. This chapter seeks to review the accomplishments attained during the completion of this research project, as well as the impacts it may have.

The project began with a literature review in order to understand the background knowledge required to complete the project. This review included research about various techniques that are required to preprocess data, and various machine learning model approaches to event detection. The project incorporated these ideas in various areas of the project, easing the overall endeavor.

A variety of functions are employed in order to properly preprocess data that is retrieved from the Trove API, these include cleaning, tokenization, and stop word removal. This data is then stored into a large MongoDB database, such that the application can easily access information without having to depend on the Trove API when querying information. This database serves as the material the application utilises in order to construct the datasets that will build the machine learning models.

Feature selection involved evaluating the usefulness of a diverse range of features, ranging from frequencies of articles to the tf-idf values of varying terms. This section relies heavily on univariate selection to evaluate features, using the Pearson standard correlation coefficient to calculate the correlation values between features and event occurrences. This process is then further refined by using the feature importance values returned during the model construction and evaluation process.

We compared and contrasted the outputs and overall functionality of different types of models using performance metrics of precision, recall, and f1 scores. The scores suggested that models trained on full datasets, as opposed to undersampled datasets provided more

tangible results. Features were culled according to their relative importance as returned by these models, this decreased result ambiguity and improved the models overall. RF, SVM, and KNN models were compared, results indicated that RF models output the most convincing results and performance metrics.

Analysis of the results output by the final model, the RF average star frequency model, shows that the application has completed the objectives outlined in section 1.1 to varying degrees of success. The final application succeeds in accepting historical data as an input, and outputs the temporal locations of natural disasters that have occurred within the said time period as proposed. The research project additionally outputs a list of probable unlisted event occurrences, each with a confidence interval representing how likely the event is to exist. Discovery of an unlisted event by the model lends credence to the claim that there may be many more similar incidents that have yet to be manually investigated.

There is much historical significance in the acknowledgement and identification of previously unknown events. Those who are interested in doing so will be able to use the information gathered from this project as a reference to expand their options. Natural catastrophe modelling companies may also be able to use this information to analyse the completeness of their historical data sets.

Chapter 9

Future Work

9.1 Potential Improvements

Researching and completing this research project has led to a number of various improvements that could be made to the project. These were not implemented due to either the lack of resources or time, and are made note of in this chapter.

This project exclusively uses the Trove API to create the database and is thus solely reliant on the Trove servers being active when building the MongoDB database. It is possible to integrate other news aggregates and fetch article information from those, although this would require amendment to the functions that parse and store articles. It would be worth doing so to allow cross validation between sources, and no longer being dependent on a single source.

As briefly discussed in chapter 4, it is possible to investigate the correlation values of every existing term. This was not done because of the lack of time and resources, there is potential here to discover terms that hold more weight in indicating the presence of events. There is also the potential to improve the rather rudimentary point system used to compute the average star frequency feature. Article ranking worth can be distributed differently, perhaps leading to better overall predictions.

This project experimented mainly with three different types of machine learning algorithms, namely RF, KNN, and SVM. Future researchers can experiment with others such as Naive Bayes and C4.5, and discover whether they produce worse, similar, or better results. Manual investigation to determine if predicted events are false positives or actual unlisted events is tedious and laborious. Future researchers should construct a function or model that assists in investigating the associated IDs. This will cut down time taken considerably.

9.2 Final Words

The project is not yet entirely complete, for this program only detects the presence of cyclone natural catastrophes. This could serve as a reference or framework for future analysis of different natural catastrophe types. This project serves as proof that it is possible to detect the presence of natural catastrophe events through historical and current news articles.

Chapter 10





























Abbreviations

GUI	Graphical User Interface
NLP	Natural Language Processing
NLTK	Natural Language Toolkit
IDE	Integrated Development Environment
SVM	Support-Vector Machine
CRF	Conditional Random Field
RF	Random Forest
API	Application Program Interface
JSON	JavaScript Object Notation
HTML	HyperText Markup Language
TF-IDF	Term Frequency - Inverse Document Frequency

Appendix A

Consultation Attendance Form

Weekly Consultation Meeting Log

Week	Supervisor's Comments (e.g. un/satisfactory performance, warning, specifics where improvement required, etc.)	Supervisor's Signature	Student's Signature	Date
1	Excellent progress: fetching of data from Traex implemented			02/08/2019
2	Excellent progress: figured out but data are not very good predictors.			09/08/2019
3	Excellent progress: learnt about limitation of modelling if we have a bad data			16/08/2019
4	Excellent progress. Got around first preliminary thesis and knows what to do next.			23/08/2019
5	e-mail update.			6/09/2019
6	Good progress. Almost done with preliminary thesis.			6/09/2019
7	Good progress - looked at other features and re-built data set.			13/09/2019
8	Good progress - hitting some issues with training data but have found possible work-around.			17/09/2019
8	Good progress - still defining confidence.			04/10/2019
9	Very good progress - solved few bugs in the implementation.			11/10/2019
10	Excellent progress - improved features.			18/10/2019
11	Excellent progress - some really positive results.			25/10/2019
12	e-mail update.			1/11/2019
13	Excellent progress - finalising some parts of the thesis			6/11/2019

Appendix B

Code

B.1 Trove API Data Retrieval Functions

```
1 import json
2 import urllib
3 import urllib.request
4 from time import sleep
5 from urllib.error import HTTPError
6
7 from backoff import on_exception, expo
8 from ratelimit import limits, RateLimitException
9
10
11 @on_exception(expo, RateLimitException, max_time=60)
12 @limits(calls=100, period=60)
13 def call_api(url):
14     """Requests response from URL. Complies with call limit.
15
16     :param String url: valid encoded url request
17     :return: response
18     :rtype: object
19     """
20     attempts = 0
21     while attempts < 10:
22         try:
23             response = urllib.request.urlopen(url)
24             return response
25         except HTTPError as e:
26             print("Encountered Error: ", e)
27             print("Attempting to reconnect...")
28             sleep(6)
29             attempts += 1
30
```

```

31 response = urllib.request.urlopen(url)
32 if response.getcode() != 200:
33     raise Exception('API response: {}'.format(response.getcode()))
34 return response
35
36
37 def url_encode(t):
38     """Encodes input string to comply with URL requirements
39
40     :param t: input string
41     :return: url encoded string
42     """
43     return urllib.request.pathname2url(t)
44
45
46 def trove_api_request(trove_key, zone, category, search_terms, min_year,
47                       max_year, s, n):
48     """Makes a search request to the Trove API with the provided
49     search_terms and returns the selected page of results in JSON format
50
51     :param trove_key:
52     :param zone:
53     :param category:
54     :param search_terms:
55     :param min_year:
56     :param max_year:
57     :param s:
58     :param n:
59     :return:
60     """
61
62     search_terms_url = url_encode(search_terms)
63
64     request = "https://api.trove.nla.gov.au/v2/result?key=" + trove_key
65     + \
66         "&zone=" + zone + \
67         "&q=" + search_terms_url + \
68         "%20date%3A%5B" + str(min_year) + \
69         "%20TO%20" + str(max_year) + \
70         "%5D&encoding=json&s=" + str(s) + \
71         "&n=" + str(n) + \
72         "&reclevel=full&include=articletext&l-australian&l-
73 category=" + category
74
75     # print(request)
76     response = call_api(request)
77     response_content = response.read()
78     response_json = json.loads(response_content)
79     return response_json
80
81
82 def trove_api_get(trove_key, article_id):

```

```
80     """Makes an article request to the Trove API with the provided
81     article_id, returning the result in JSON format
82
83     :param trove_key:
84     :param article_id:
85     :return:
86     """
87
88     request = "https://api.trove.nla.gov.au/v2/newspaper/" + str(
89         article_id) + "?key=" + trove_key + "&include=articletext&
90     encoding=json&reclevel=full&include=articletext"
91     response = call_api(request)
92     response_content = response.read()
93     print(request)
94
95     try:
96         response_json = json.loads(response_content)
97         return response_json
98     except:
99         raise Exception
```

B.2 Article Cleaning, Preprocessing, and Storage

```
1  import csv
2  import string
3  from datetime import datetime
4
5  import pandas as pd
6  from bs4 import BeautifulSoup
7  from nltk import FreqDist
8  from nltk.corpus import stopwords, words
9  from nltk.stem import *
10 from nltk.tokenize import *
11 from sklearn.feature_extraction.text import TfidfVectorizer
12
13 from Detector import troveAPI
14
15
16 def read_csv(filename):
17     """Reads csv file, not inputting any with missing date information
18     up to day i.e YYYY/MM/DD, hour is not recorded
19
20     :param filename:
21     :return peril_list: A list of dictionaries with peril and date in
22     datetime format
23     """
24     reader = csv.DictReader(open(filename))
25     peril_list = []
```

```

24
25     for row in reader:
26         if row['Start Year'] == 'NULL' or row['Start Year'] == '0' or \
27             row['Start Month'] == 'NULL' or row['Start Month'] == '0'
28         ' or \
29             row['Start Day'] == 'NULL' or row['Start Day'] == '0':
30             continue
31
32         peril = row['Peril Type']
33         date = row['Start Year'] + '-' + row['Start Month'] + '-' + row[
34             'Start Day']
35         try:
36             date = datetime.strptime(date, '%Y-%m-%d')
37             peril_dict = {'peril': peril,
38                           'date': date}
39             peril_list.append(peril_dict)
40         except ValueError as e:
41             print(e)
42             continue
43
44     return peril_list
45
46 def pre_process(text):
47     """Pre-process/clean text. Removing markup, reg. expressions/
48     punctuation, stop words, then tokenizing and applying lemmatization.
49
50     :param text:
51     :return: Text without stopwords, punctuation, markup and all
52     lemmatized.
53     """
54
55     # Remove Markup
56     soup = BeautifulSoup(text, features="html.parser")
57
58     for script in soup(["script", "style"]):
59         script.extract()
60
61     text = soup.get_text()
62
63     # Remove reg. ex. / punctuation, this also removes the hyphen in
64     hyphenated words i.e freeze-dry -> freeze dry
65     text = re.sub(r'^\w', ' ', text)
66
67     # Tokenize and transform into lower case
68     text = word_tokenize(text)
69     text = [w.lower() for w in text]
70
71     # Remove stop words
72     english_words = set(words.words())
73     stop_words = set(stopwords.words('english'))
74     newstopwords = ['tho', 'mr', 'tbe', '000']

```

```

71     stop_words.update(newstopwords)
72     filtered_text = [w for w in text if
73                     w.lower() in english_words and w.lower() not in
74                     stop_words and w not in string.punctuation and len(
75                         w) > 2]
76
77     # Lemmatisation
78     lemmatizer = WordNetLemmatizer()
79     lemmatized_text = ' '.join(lemmatizer.lemmatize(token) for token in
80     filtered_text)
81     lemmatized_tokenized_text = word_tokenize(lemmatized_text)
82
83     return lemmatized_tokenized_text
84
85 def compute_term_frequency(tokenized_text):
86     """Calculate term count, term frequency and term frequency versus
87     term count.
88
89     :param tokenized_text:
90     :return: return frequency of words and frequency of words against
91     amount of words
92     """
93     # Frequency distribution
94     term_count = len(tokenized_text)
95     i = 0
96     frequency_distribution = FreqDist(tokenized_text)
97     tf = []
98     while i < len(frequency_distribution):
99         # Where [i][1] is the frequency of the word
100         tf.append((frequency_distribution.most_common()[i][0],
101                   frequency_distribution.most_common()[i][1] / term_count))
102         i = i + 1
103     return term_count, frequency_distribution, tf
104
105 def process_text(trove_key, article_id):
106     """UNUSED REDUNDANT function that fetches articletext given article
107     ID and processes it. Pre-process and process key and article, grabs
108     article text then processes it.
109
110     :param article_id:
111     :param trove_key:
112     :return: list of most frequent tokens
113     """
114     data = troveAPI.trove_api_get(trove_key, article_id)
115     text = data['article']['articleText']
116     processed_text = pre_process(text)
117     return processed_text
118
119 def process_search(trove_key, zone, category, search_terms, min_year,
120                   max_year, s, n):

```

```

115     """Function that makes search requests to Trove. This allows every
116     result to be processed. We now use a while loop instead of a
117     recursive loop, as there is no limitation on iterations.
118
119     :param trove_key:
120     :param zone:
121     :param category:
122     :param search_terms:
123     :param min_year:
124     :param max_year:
125     :param s: The key that goes to the next page of search results
126     :param n: Number of results listed per page
127
128     :return:
129     """
130     data = troveAPI.trove_api_request(trove_key, zone, category,
131     search_terms, min_year, max_year, s, n)
132     total = data['response']['zone'][0]['records']['total']
133     print(total, 'items found')
134     skip_year = 0
135     year_skipped = False
136     skip_pages = 0
137     old_pages = 0
138     pages = 0
139     count = 0
140     article_collection = []
141     next_exists = True
142
143     # Keep looping as long as there is another page to be processed
144     while next_exists:
145         if skip_pages == 0:
146             i = 0
147             record_size = data['response']['zone'][0]['records']['n']
148             record_size = int(record_size)
149             s = data['response']['zone'][0]['records']['nextStart']
150
151             # Construction of the dictionary to be appended to the list
152             while i < record_size and skip_pages == 0:
153                 try:
154                     article_id = data['response']['zone'][0]['records']['
155                     'article'][i]['id']
156                     processed_text = pre_process(data['response']['zone'
157                     ][0]['records']['article'][i]['articleText'])
158                     date = data['response']['zone'][0]['records']['
159                     article'][i]['date']
160                     date = datetime.strptime(date, '%Y-%m-%d')
161                     term_count, word_frequency, term_frequency =
162                     compute_term_frequency(processed_text)
163                     article_dictionary = {"date": date,
164                                         "id": article_id,
165                                         "processed text":
166                     processed_text,

```

```

159         "term count": term_count,
160         "word frequency":
word_frequency,
161         "term frequency":
term_frequency,
162     }
163     article_collection.append(article_dictionary)
164     i += 1
165     count += 1
166     except KeyError as e:
167         print('Could not find key', e)
168         print('Skipping this article...')
169         i += 1
170         count += 1
171     print(count, 'articles processed')
172 else:
173     skip_pages -= 1
174
175     pages += 1
176     data = troveAPI.trove_api_request(trove_key, zone, category,
search_terms, min_year, max_year, s, n)
177     total = data['response']['zone'][0]['records']['total']
178     total = int(total)
179
180     if skip_year >= 15:
181         print('This year will be skipped, too many non-results
returned from API!')
182         year_skipped = True
183         next_exists = False
184
185     # If we encounter an error resulting in an empty page of results
during another restart
186     if total == 0 and skip_pages != 0:
187         print('Error encountered during restart, empty page of
results returned... Restarting search from scratch')
188         if pages > old_pages - 3:
189             skip_year += 1
190             skip_pages = old_pages
191             pages = 0
192             s = '*'
193             data = troveAPI.trove_api_request(trove_key, zone, category,
search_terms, min_year, max_year, s, n)
194             next_exists = True
195     # For the first time we encounter the error, or when it occurs
on an unexplored page
196     elif total == 0 and skip_pages == 0:
197         print('Error encountered, empty page of results returned...
Restarting search from scratch')
198         if old_pages == pages:
199             skip_year += 1
200             print('Same restart happened:', skip_year, 'times')
201     else:

```

```

202         skip_year = 0
203         old_pages = pages
204         skip_pages = pages
205         pages = 0
206         s = '*'
207         data = troveAPI.trove_api_request(trove_key, zone, category,
search_terms, min_year, max_year, s, n)
208         next_exists = True
209         # There is no error encountered
210         elif 'nextStart' in data['response']['zone'][0]['records']:
211             if skip_pages != 0:
212                 print('Skipping page: ', pages, '...', sep='')
213                 s = data['response']['zone'][0]['records']['nextStart']
214             else:
215                 next_exists = False
216
217         return article_collection, year_skipped
218
219
220 def identity_tokenizer(text):
221     """Necessary function for tfidf to work
222
223     :param text:
224     :return:
225     """
226     return text
227
228
229 def tfidf_func(data, total):
230     """Inputting a list of tokenized texts, performs sklearn's tf-idf
function. Prints a list of tf-idf values in comparison to the first
document
231     Code extracted from https://kavita-ganesan.com/tfidftransformer-tfidfvectorizer-usage-differences/. Some minor adjustments made such
that the TfidfVectorizer accepts tokenized inputs.
232
233     :param total: total amount of articles being processed
234     :param data
235     :return:
236     """
237     # settings that you use for count vectorizer will go here
238     tfidf_vectorizer = TfidfVectorizer(tokenizer=identity_tokenizer,
lowercase=False)
239     # just send in all your docs here
240     tfidf_vectorizer_vectors = tfidf_vectorizer.fit_transform(data)
241     # print(tfidf_vectorizer_vectors.shape)
242     # get the first vector out (for the first document)
243     first_vector_tfidfvectorizer = tfidf_vectorizer_vectors[0]
244
245     # place tf-idf values in a pandas data frame
246     df = pd.DataFrame(first_vector_tfidfvectorizer.T.todense(), index=
tfidf_vectorizer.get_feature_names(),

```



```

247         columns=[0])
248
249     # iterate over each article to store into data frame
250     i = 1
251     int_total = int(total)
252     while i < int_total:
253         df[i] = tfidf_vectorizer_vectors[i].T.todense()
254         i += 1
255
256     return df

```

B.3 MongoDB Retrieval and Dataset Construction

```

1  import time
2  from datetime import timedelta, datetime
3  import pandas as pd
4  import pymongo
5  from pandas.plotting import register_matplotlib_converters
6
7  from Detector.articleProcessing import (read_csv, process_search,
8      tfidf_func)
9
10 register_matplotlib_converters()
11
12 def store_csv_data(filename):
13     """ Input filename, read and process csv and then store into MongoDB
14         database
15
16     :param filename:
17     :return:
18     """
19     client = pymongo.MongoClient("mongodb://localhost:27017/")
20     db = client['perilAUS']
21     col = db['events']
22     data = read_csv(filename)
23     col.insert_many(data)
24
25 def extract_and_insert(trove_key, search_terms, start_year, end_year,
26     database_name, collection_name):
27     """ Extract information from the Trove API, process it, and insert
28         it into the MongoDB database
29
30     :param collection_name: name of collection information is being
31         stored to
32     :param database_name: name of database information is being stored
33         to

```

```

30 :param end_year:
31 :param start_year:
32 :param search_terms:
33 :param trove_key
34
35 :return:
36 """
37 i = 0
38 client = pymongo.MongoClient("mongodb://localhost:27017/")
39 db = client[database_name]
40 col = db[collection_name]
41
42 skipped_years = []
43
44 while i <= (int(end_year) - int(start_year)):
45     year = int(start_year) + i
46
47     print('Collecting articles from year:', year)
48     art_col, skipped = process_search(trove_key, 'newspaper', '
Article&bulkHarvest=true', search_terms, year,
49                                     year, '*',
50                                     100)
51     print(len(art_col), 'items inserted into database:',
database_name, 'and into collection:', collection_name)
52
53     inserted_list = art_col
54     col.insert_many(inserted_list)
55
56     if skipped:
57         skipped_years.append(year)
58     i += 1
59
60 print('\t'.join(map(str, skipped_years)))
61 return
62
63
64 def get_data_between_dates(begin_date, end_date, database_name,
collection_name):
65     """ Get data between specified dates from MongoDB
66
67     :param collection_name: name of collection information is being
stored to
68     :param database_name: name of database information is being stored
to
69     :param begin_date:
70     :param end_date:
71
72     :return:
73     """
74     client = pymongo.MongoClient("mongodb://localhost:27017/")
75     db = client[database_name]
76     col = db[collection_name]

```

```

77     if isinstance(begin_date, str):
78         begin_date = datetime.strptime(begin_date, '%Y-%m-%d')
79         end_date = datetime.strptime(end_date, '%Y-%m-%d')
80     data = []
81     query = col.find({'date': {'$lt': end_date, '$gt': begin_date}})
82
83     for item in query:
84         data.append(item)
85
86     return data
87
88
89 def delete_data_between_dates(begin_date, end_date, database_name,
90                               collection_name):
91     """ Delete all records in specified date range
92
93     :param begin_date:
94     :param end_date:
95     :param database_name:
96     :param collection_name:
97     :return:
98     """
99     client = pymongo.MongoClient("mongodb://localhost:27017/")
100    db = client[database_name]
101    col = db[collection_name]
102
103    if isinstance(begin_date, str):
104        begin_date = datetime.strptime(begin_date, '%Y-%m-%d')
105        end_date = datetime.strptime(end_date, '%Y-%m-%d')
106
107    query = col.delete_many({'date': {'$lt': end_date, '$gt': begin_date
108    }})
109
110    print(query.deleted_count, "docs deleted")
111
112    return
113
114 def process_data(data):
115     """ Input tokenized textual data and output tfidf scores
116
117     :param data:
118     :return:
119     """
120     processed_text = []
121     for article in data:
122         processed_text.append(article['processed text'])
123
124     if len(data) == 0:
125         print(processed_text)
126     return tfidf_func(processed_text, len(data))

```

```

127
128 def get_tfidf_in_proximity(begin_date, end_date, database_name,
129                             collection_name, proximity):
130     """ Collates tf-idf values over a specified range, in chunks of size
131         relative to proximity value. Returns a list of dataframes containing
132         average tf-idf values of words.
133
134     :param begin_date:
135     :param end_date:
136     :param database_name:
137     :param collection_name:
138     :param proximity:
139     :return:
140     """
141
142     func_start = time.time()
143     if isinstance(begin_date, str):
144         begin_date = datetime.strptime(begin_date, '%Y-%m-%d')
145         end_date = datetime.strptime(end_date, '%Y-%m-%d')
146     delta = end_date - begin_date
147
148     i = 0
149     data = []
150     while i < delta.days + proximity:
151         from_date = begin_date + timedelta(days=i)
152         to_date = from_date + timedelta(days=proximity)
153         data_to_be_processed = get_data_between_dates(from_date, to_date
154 , database_name, collection_name)
155         if data_to_be_processed:
156             df = process_data(data_to_be_processed)
157             df['avg'] = df.mean(axis=1)
158             df = df.avg
159             data.append(df)
160         else:
161             data.append(None)
162         i += 1
163
164     func_end = time.time()
165     print('Seconds elapsed during tfidf value collection:', func_end -
166 func_start)
167
168     return data
169
170 def get_frequency_per_day(begin_date, end_date, database_name,
171                             collection_name, proximity_overflow):
172     """ Get information from MongoDB database and gather frequency of
173         articles occurrence per day
174
175     :param proximity_overflow:
176     :param collection_name: name of collection information is being
177         stored to

```

```

171     :param database_name: name of database information is being stored
    to
172     :param begin_date:
173     :param end_date:
174
175     :return:
176     """
177     client = pymongo.MongoClient("mongodb://localhost:27017/")
178     db = client[database_name]
179     col = db[collection_name]
180
181     if isinstance(begin_date, str):
182         begin_date = datetime.strptime(begin_date, '%Y-%m-%d')
183         end_date = datetime.strptime(end_date, '%Y-%m-%d')
184     delta = end_date - begin_date
185     i = 0
186     data = []
187
188     while i < delta.days + proximity_overflow:
189         check_date = begin_date + timedelta(days=i)
190         count = col.count_documents({"date": check_date})
191
192         date_data = [check_date, count]
193         data.append(date_data)
194         i += 1
195
196     return data
197
198
199 def get_word_frequency_per_day(begin_date, end_date, database_name,
    collection_name, proximity_overflow):
200     """ Similar to getting article frequency per day, but grabs from the
    already stored 'word frequency' and 'term frequency' columns, which
    hold word frequency, and word frequency over article length.
201     Seen words are stored in a list, and if the word is seen again,
    instead of appending another value to the list, it adds to the
    already existing value.
202
203     :param begin_date:
204     :param end_date:
205     :param database_name:
206     :param collection_name:
207     :param proximity_overflow:
208     :return:
209     """
210     client = pymongo.MongoClient("mongodb://localhost:27017/")
211     db = client[database_name]
212     col = db[collection_name]
213
214     func_start = time.time()
215
216     if isinstance(begin_date, str):

```

```

217     begin_date = datetime.strptime(begin_date, '%Y-%m-%d')
218     end_date = datetime.strptime(end_date, '%Y-%m-%d')
219     delta = end_date - begin_date
220     i = 0
221     data = []
222
223     while i < delta.days + proximity_overflow:
224         word_seen = []
225         frequency_dict = {}
226         frequency_over_article_length = {}
227         check_date = begin_date + timedelta(days=i)
228         query = col.find({"date": check_date})
229         for article in query:
230             j = 0
231             for word in article['word frequency']:
232                 if word in word_seen:
233                     frequency_dict[word] += article['word frequency'].
get(word, "")
234                     frequency_over_article_length[word] += article['term
frequency'][j][1]
235                 else:
236                     word_seen.append(word)
237                     frequency_dict[word] = article['word frequency'].get
(word, "")
238                     frequency_over_article_length[word] = article['term
frequency'][j][1]
239
240             j += 1
241
242         date_data = [check_date, frequency_dict,
frequency_over_article_length]
243         data.append(date_data)
244         i += 1
245         # print('day:', i)
246
247     func_end = time.time()
248     print('Seconds elapsed during word frequency collection:', func_end
- func_start)
249     return data
250
251
252 def get_rankings_per_day(begin_date, end_date, database_name,
collection_name, proximity_overflow):
253
254     client = pymongo.MongoClient("mongodb://localhost:27017/")
255     db = client[database_name]
256     col = db[collection_name]
257     if isinstance(begin_date, str):
258         begin_date = datetime.strptime(begin_date, '%Y-%m-%d')
259         end_date = datetime.strptime(end_date, '%Y-%m-%d')
260     delta = end_date - begin_date
261     i = 0

```

```

262     data = []
263
264     while i < delta.days + proximity_overflow:
265         check_date = begin_date + timedelta(days=i)
266         query = col.find({"date": check_date})
267         one_star, two_star, three_star, four_star, five_star = 0, 0, 0,
0, 0
268         for article in query:
269             rating = article['auto_rating']
270             if rating == 1:
271                 one_star += 1
272             elif rating == 2:
273                 two_star += 1
274             elif rating == 3:
275                 three_star += 1
276             elif rating == 4:
277                 four_star += 1
278             elif rating == 5:
279                 five_star += 1
280         rating_dict = {
281             'one': one_star,
282             'two': two_star,
283             'three': three_star,
284             'four': four_star,
285             'five': five_star
286         }
287         data.append(rating_dict)
288         i += 1
289
290     return data
291
292
293 def get_peril_dates(peril_name, begin_date, end_date, database_name,
collection_name):
294     """ Grab data from MongoDB using previously created function then
put dates of event occurrences in a list
295
296     :param peril_name:
297     :param begin_date:
298     :param end_date:
299     :param database_name:
300     :param collection_name:
301     :return:
302     """
303
304     peril_data = get_data_between_dates(begin_date, end_date,
database_name, collection_name)
305     peril_list = []
306     for item in peril_data:
307         if item['peril'] == peril_name:
308             peril_date = item['date']
309             peril_list.append(peril_date)

```

```

310
311     return peril_list
312
313
314 def create_ranker_df(begin_date, end_date, database_name,
315 collection_name):
316     """Create the dataframe input for LucyBot's ranker function
317
318     :param begin_date:
319     :param end_date:
320     :param database_name:
321     :param collection_name:
322     :return:
323     """
324
325     data = get_data_between_dates(begin_date, end_date, database_name,
326 collection_name)
327     df_list = []
328     i = 0
329     for item in data:
330         if not item.get('processed text'):
331             print('empty list found, skipping')
332         if len(item.get('processed text')) < 5:
333             print('list less than five found')
334         else:
335             tokenized_text = item.get('processed text')
336             joined_text = ' '.join(tokenized_text)
337             item['id'] = item.get('id')
338             item['articleText'] = joined_text
339             df_list.append(item)
340             i += 1
341
342     df = pd.DataFrame(df_list)
343     return df
344
345
346 def filter_and_insert(begin_date, end_date, database_name,
347 collection_name, insert_database, insert_collection):
348     client = pymongo.MongoClient("mongodb://localhost:27017/")
349     db = client[insert_database]
350     col = db[insert_collection]
351     from play_ground import filter_articles
352
353     df = filter_articles(begin_date, end_date, database_name,
354 collection_name)
355
356     article_list = df.to_dict('records')
357     col.insert_many(article_list)
358
359     return

```


B.4 Feature Selection

```

1 import time
2 from datetime import timedelta
3 import os
4 import matplotlib.pyplot as plt
5 import pandas as pd
6 from sklearn import preprocessing
7 from statsmodels.formula.api import ols
8
9 from Detector.database import (get_data_between_dates, get_peril_dates,
    process_data, get_word_frequency_per_day,
10                                get_frequency_per_day,
    get_tfidf_in_proximity, get_rankings_per_day)
11
12
13 def compile_values(peril_data, article_frequency_data,
    term_frequency_data, proximity_range, keyword, weighted_values):
14     """ OLD function, no longer in use due to little modularisation and
    flexibility.
15     Grab values from article freq data and word freq data, and put them
    together, such that accessing them and inputting them into a Data
    frame is easier.
16
17     :param peril_data:
18     :param article_frequency_data:
19     :param term_frequency_data:
20     :param proximity_range:
21     :param keyword:
22     :param weighted_values:
23     :return:
24     """
25     day_weight = 1
26     event_occurrence = False
27     for i in range(0, (len(article_frequency_data) - proximity_range)):
28         proximity = 0
29         try:
30             article_frequency_data[i].append(float(int(
    term_frequency_data[i][1].get(keyword, ""))))
31             article_frequency_data[i].append(term_frequency_data[i][2].
    get(keyword, ""))
32         except (IndexError, ValueError):
33             article_frequency_data[i].append(0)
34             article_frequency_data[i].append(0)
35         while proximity < proximity_range:
36             weight = 1
37             if weighted_values:
38                 weight = 1 - ((proximity_range - proximity) /
    proximity_range * 0.1)
39             article_frequency_data[i][1] += (article_frequency_data[i +
    proximity + 1][1] * weight)

```

```

40
41         try:
42             article_frequency_data[i][2] += float(int(
term_frequency_data[i + proximity + 1][1].get(keyword, ""))
43             article_frequency_data[i][3] += term_frequency_data[i +
proximity + 1][2].get(keyword, "")
44         except (IndexError, ValueError):
45             article_frequency_data[i][2] += 0
46             article_frequency_data[i][3] += 0
47             proximity += 1
48
49         if day_weight == 30:
50             event_occurrence = False
51         if article_frequency_data[i][0] in peril_data:
52             event_occurrence = True
53             day_weight = 1
54             article_frequency_data[i].append(0)
55         elif event_occurrence:
56             article_frequency_data[i].append(day_weight)
57             day_weight += 1
58         else:
59             article_frequency_data[i].append(100)
60
61         del article_frequency_data[(len(article_frequency_data) -
proximity_range):]
62         print(article_frequency_data)
63         return article_frequency_data
64
65
66 def correlate_frequency_event_occurrence(peril_name, proximity_range,
begin_date, end_date, peril_database,
67                                         peril_collection,
68                                         article_database, article_collection, keyword,
69                                         weighted_values=True, plot=
False):
70     """ OLD function, no longer in use due to little modularisation and
flexibility.
71     Get frequency per day and event occurrence, put them into a
DataFrame and correlate. Also produce a plot showing both.
72
73     :param plot:
74     :param keyword:
75     :param weighted_values:
76     :param peril_name:
77     :param proximity_range:
78     :param begin_date:
79     :param end_date:
80     :param peril_database:
81     :param peril_collection:
82     :param article_database:
83     :param article_collection:
84     :return:

```

```

84     """
85     peril_data = get_peril_dates(peril_name, begin_date, end_date,
86     peril_database, peril_collection)
87     article_data = get_frequency_per_day(begin_date, end_date,
88     article_database, article_collection, proximity_range)
89     word_frequency_data = get_word_frequency_per_day(begin_date,
90     end_date, article_database, article_collection,
91     proximity_range)
92     data = compile_values(peril_data, article_data, word_frequency_data,
93     proximity_range, keyword, weighted_values)
94
95     func_start = time.time()
96     df = pd.DataFrame(data)
97     scaler = preprocessing.MinMaxScaler()
98     print(df)
99     df = df[df[4] != 100]
100    print(df)
101    df[[1, 2, 3, 4]] = scaler.fit_transform(df[[1, 2, 3, 4]])
102    print(df)
103
104    correlation_val = df[1].corr(df[4])
105    correlation_val2 = df[2].corr(df[4])
106    correlation_val3 = df[3].corr(df[4])
107    func_end = time.time()
108    print('Seconds elapsed during standardization:', func_end -
109    func_start)
110
111    df.columns = ['date', 'art_freq', 'word_freq', 'word_len_freq', '
112    event']
113    print(df.corr())
114    model = ols("event ~ art_freq", data=df).fit()
115    print(model.params)
116    print(model.summary())
117
118    if plot:
119        ax = plt.gca()
120        df.plot(x=0, y=1, label='Article Frequency in Range', ax=ax)
121        df.plot(x=0, y=2, label='Word Frequency in Range', ax=ax)
122        df.plot(x=0, y=3, label='Word Frequency over Article Length in
123        Range', ax=ax)
124        df.plot(x=0, y=4, label='Event Occurrence', ax=ax)
125        plt.xlabel('Date')
126        plt.ylabel('Frequency')
127        plt.legend(loc='upper right')
128        plt.show()
129
130    print(correlation_val, 'with a proximity range of', proximity_range)
131    print(correlation_val2, 'with a proximity range of', proximity_range)
132    print(correlation_val3, 'with a proximity range of', proximity_range)

```

```

127     # change date from column value to index for later training and
128     testing
129     df.set_index('date', inplace=True)
130
131     return df
132
133 def gather_training_data(peril_name, proximity_range, begin_date,
134     end_date, peril_database,
135     peril_collection, article_database,
136     article_collection):
137     """Gathers data from MongoDB, and inserts them into one single list
138     so that accessing is easier
139
140     :param peril_name:
141     :param proximity_range:
142     :param begin_date:
143     :param end_date:
144     :param peril_database:
145     :param peril_collection:
146     :param article_database:
147     :param article_collection:
148     :return:
149     """
150     peril_data = get_peril_dates(peril_name, begin_date, end_date,
151     peril_database, peril_collection)
152
153     data = []
154     print(len(peril_data), 'perils found')
155
156     for item in peril_data:
157         peril_start = item
158         peril_end = item + timedelta(days=30)
159         article_data = get_frequency_per_day(peril_start, peril_end,
160         article_database, article_collection,
161         proximity_range)
162         word_frequency_data = get_word_frequency_per_day(peril_start,
163         peril_end, article_database, article_collection,
164         proximity_range)
165         tfidf_data = get_tfidf_in_proximity(peril_start, peril_end,
166         article_database, article_collection,
167         proximity_range)
168         ranking_data = get_rankings_per_day(peril_start, peril_end,
169         article_database, article_collection,
170         proximity_range)
171
172         i = 0
173         for element in article_data:
174             try:
175                 freq = {
176                     'date': element[0],
177                     'article_freq': element[1],

```



```

213     peril_data = get_peril_dates(peril_name, begin_date, end_date,
214     peril_database, peril_collection)
215     ranking_data = get_rankings_per_day(begin_date, end_date,
216     article_database, article_collection,
217     proximity_range)
218
219     i = 0
220     day_value = 0
221     none_added = 0
222     event_occurrence = False
223
224     for element in article_data:
225         if day_value == 30:
226             event_occurrence = False
227         if element[0] in peril_data:
228             event_occurrence = True
229             day_value = 0
230         elif event_occurrence:
231             day_value += 1
232         else:
233             day_value = 100
234         try:
235             tfidf = tfidf_data[i]
236         except IndexError as e:
237             print('tfidf missing val', e)
238         try:
239             freq = {
240                 'date': element[0],
241                 'article_freq': element[1],
242                 'word_freq': word_frequency_data[i][2],
243                 'tfidf_val': tfidf_data[i],
244                 'ranks': ranking_data[i],
245                 'event_value': day_value
246             }
247             data.append(freq)
248         except IndexError:
249             print(element[1], 'articles found on', element[0], '
250             inserting a None row')
251             freq = {
252                 'date': element[0],
253                 'article_freq': element[1],
254                 'word_freq': word_frequency_data[i][2],
255                 'tfidf_val': tfidf_data[i],
256                 'ranks': ranking_data[i],
257                 'event_value': day_value
258             }
259             data.append(freq)
260             none_added += 1
261             i += 1
262     print('None rows added:', none_added)
263
264     return data

```

```

262
263
264 def build_dataset(peril_name, proximity_range, begin_date, end_date,
265                  peril_database,
266                  peril_collection, article_database, article_collection
267                  , keyword_list, except_keywords,
268                  weighted_values=True, plot=False, filename=None, train
269                  =False):
270     """This is the main function that builds our dataset. It returns our
271     dependent variable and features in a single dataframe indexed by
272     date.
273     Also returns correlation values of features and plots data if
274     specified to. Can also optionally plot, and save the final dataframe.
275
276     :param train:
277     :param filename:
278     :param peril_name:
279     :param proximity_range:
280     :param begin_date:
281     :param end_date:
282     :param peril_database:
283     :param peril_collection:
284     :param article_database:
285     :param article_collection:
286     :param keyword_list:
287     :param weighted_values:
288     :param plot:
289     :return:
290     """
291     if train is True:
292         data = gather_training_data(peril_name, proximity_range,
293                                     begin_date, end_date, peril_database, peril_collection,
294                                     article_database, article_collection
295 )
296         print('data length is', len(data))
297     else:
298         data = gather_all_data(peril_name, proximity_range, begin_date,
299                               end_date, peril_database, peril_collection,
300                               article_database, article_collection)
301         print('data length is', len(data))
302
303     date_list = []
304     event_list = []
305     article_frequency_list = create_feature_article_frequency(data,
306                                                                proximity_range, weighted_values)
307
308     for item in data:
309         date_list.append(item['date'])
310         event_list.append(item['event_value'])
311
312     df = pd.DataFrame({'date': date_list})
313     df['event'] = event_list

```

```

304 df.drop(df.tail(proximity_range).index, inplace=True)
305 df['article frequency'] = article_frequency_list
306
307 for keyword in keyword_list:
308     header = ('' + keyword + '')
309     if keyword in except_keywords:
310         word_frequency_list = create_feature_word_freq(data,
311 proximity_range, keyword, weighted_values)
312         df[header + ' frequency'] = word_frequency_list
313         tfidf_list = create_feature_word_tfidf(data, proximity_range,
314 keyword, weighted_values)
315         df[header + ' tfidf value'] = tfidf_list
316
317 one_list, two_list, three_list, four_list, five_list, average_list =
318 create_feature_rankings(data, proximity_range, weighted_values)
319 # df['one star frequency'] = one_list
320 # df['two star frequency'] = two_list
321 # df['three star frequency'] = three_list
322 # df['four star frequency'] = four_list
323 df['five star frequency'] = five_list
324 df['avg star frequency'] = average_list
325
326 df = normalize_df(df, train)
327 print(df.corr())
328
329 if plot:
330     plot_df(df)
331
332 df.set_index('date', inplace=True)
333 if filename is not None:
334     path = r'C:\Users\New User\PycharmProjects\engg460-research-
335 project\Data'
336     path = os.path.join(path, filename)
337     df.to_pickle(path)
338
339 return df
340
341 def plot_df(df, load=False):
342     """Plot the whole of the dataset, works only specific to the
343     dataframe built in build_dataset.
344     Added parameter incase we want to plot an already saved dataframe.
345
346     :param load:
347     :param df:
348     :return:
349     """
350     if load:
351         df.reset_index(level=0, inplace=True)
352     column_list = list(df)
353     index = column_list[0]

```



```

351     column_list = column_list[1:]
352     df.plot(x=index, y=column_list)
353     plt.xlabel('Date')
354     plt.ylabel('Frequency')
355     plt.legend(loc='upper right')
356     plt.grid()
357     plt.show()
358
359     return
360
361
362 def normalize_df(df, train=False):
363     """Normalizes the dataframe with a minmax scaler, such that all
364     values lie within range 0 - 1.
365     Achieves this by first indexing date, then resetting it (as it
366     breaks normalization).
367     Additional argument for whether it is for training data or not, to
368     remove any data outside of range
369
370     :param train:
371     :param df:
372     :return:
373     """
374     if train:
375         df = df[df['event'] <= 29]
376         df.set_index('date', inplace=True)
377         normalized_df = (df - df.min()) / (df.max() - df.min())
378         normalized_df = normalized_df.loc[~normalized_df.index.duplicated(
379             keep='last')]
380         normalized_df.reset_index(level=0, inplace=True)
381         return normalized_df
382
383
384 def create_feature_article_frequency(data, proximity_range,
385     weighted_values=True):
386     """ Creates a list of article frequencies corresponding to dates.
387
388     :param data:
389     :param proximity_range:
390     :param weighted_values:
391     :return:
392     """
393     article_frequency = []
394     i = 0
395     while i < (len(data) - proximity_range):
396         proximity = 0
397         article_freq_value = 0
398         weight = 1
399
400         while proximity < proximity_range:
401             if weighted_values:

```

```

397         weight = 1 * ((proximity_range - proximity) /
proximity_range)
398
399         article_freq_value += (data[i + proximity]['article_freq'] *
weight)
400         proximity += 1
401
402         article_frequency.append(article_freq_value)
403         i += 1
404
405     return article_frequency
406
407
408 def create_feature_word_freq(data, proximity_range, keyword,
weighted_values=True):
409     """Creates a list of word frequencies according to chosen keyword
corresponding to dates.
410
411     :param data:
412     :param proximity_range:
413     :param keyword:
414     :param weighted_values:
415     :return:
416     """
417     word_frequency = []
418     i = 0
419     zeroes_added = 0
420     while i < (len(data) - proximity_range):
421         proximity = 0
422         word_freq_value = 0
423         weight = 1
424
425         while proximity < proximity_range:
426             if weighted_values:
427                 weight = 1 * ((proximity_range - proximity) /
proximity_range)
428                 if data[i + proximity]['word_freq'] is None:
429                     freq_val = 0
430                     zeroes_added += 1
431                 else:
432                     freq_val = data[i + proximity]['word_freq'].get(keyword,
"")
433                     if isinstance(freq_val, str):
434                         freq_val = 0
435                         zeroes_added += 1
436                     word_freq_value += (freq_val * weight)
437                     proximity += 1
438
439                 word_frequency.append(word_freq_value)
440                 i += 1
441         print(i, 'items processed for word freq')
442         print(zeroes_added, 'zeroes added')

```

```

443     return word_frequency
444
445
446 def create_feature_word_tfidf(data, proximity_range, keyword,
447                               weighted_values=True):
448     """Creates a list of average tf-idf values for the keyword
449     corresponding to dates.
450
451     :param data:
452     :param proximity_range:
453     :param keyword:
454     :param weighted_values:
455     :return:
456     """
457     tfidf_list = []
458     i = 0
459     while i < (len(data) - proximity_range):
460         proximity = 0
461         tfidf_val_over_range = 0
462         weight = 1
463
464         while proximity < proximity_range:
465             if weighted_values:
466                 weight = 1 * ((proximity_range - proximity) /
467                               proximity_range)
468             try:
469                 if data[i + proximity]['tfidf_val'] is None:
470                     tfidf_val_over_range += 0
471                 else:
472                     tfidf_val = data[i + proximity]['tfidf_val'].loc[
473                         keyword]
474                     tfidf_val_over_range += (tfidf_val * weight)
475                 proximity += 1
476             except KeyError:
477                 tfidf_val_over_range += 0
478                 proximity += 1
479
480             tfidf_list.append(tfidf_val_over_range)
481             i += 1
482         return tfidf_list
483
484
485 def create_feature_rankings(data, proximity_range, weighted_values=True)
486 :
487     """
488
489     :param data:
490     :param proximity_range:
491     :param weighted_values:
492     :return:
493     """

```

```

490 one_list = []
491 two_list = []
492 three_list = []
493 four_list = []
494 five_list = []
495 average_list = []
496 i = 0
497 while i < (len(data) - proximity_range):
498     one_star, two_star, three_star, four_star, five_star = 0, 0, 0,
0, 0
499     proximity = 0
500     weight = 1
501
502     while proximity < proximity_range:
503         if weighted_values:
504             weight = 1 * ((proximity_range - proximity) /
proximity_range)
505             day_ranks = data[i + proximity]['ranks']
506             one_star += day_ranks['one'] * weight
507             two_star += day_ranks['two'] * weight
508             three_star += day_ranks['three'] * weight
509             four_star += day_ranks['four'] * weight
510             five_star += day_ranks['five'] * weight
511
512             proximity += 1
513
514             average_rank = ((one_star * 1) + (two_star * 2) + (three_star *
3) + (four_star * 4) + (five_star * 5)) / 5
515             one_list.append(one_star)
516             two_list.append(two_star)
517             three_list.append(three_star)
518             four_list.append(four_star)
519             five_list.append(five_star)
520             average_list.append(average_rank)
521
522             i += 1
523     return one_list, two_list, three_list, four_list, five_list,
average_list
524
525
526 def find_important_keywords(peril_name, proximity_range, begin_date,
end_date, peril_database,
527                             peril_collection, article_database,
article_collection):
528     """Looks at the 14 days after every peril and finds the average tf-
idf values of keywords according to that corpus
529
530     :param peril_name:
531     :param proximity_range:
532     :param begin_date:
533     :param end_date:
534     :param peril_database:

```

```

535     :param peril_collection:
536     :param article_database:
537     :param article_collection:
538     :return:
539     """
540     peril_data = get_peril_dates(peril_name, begin_date, end_date,
541     peril_database, peril_collection)
542     data = []
543     print(len(peril_data), 'perils found')
544     for item in peril_data:
545         article_data = get_data_between_dates(item, item + timedelta(
546             proximity_range), article_database,
547             article_collection)
548         # print(len(article_data), article_data[0]['date'])
549         data.extend(article_data)
550
551     df = process_data(data)
552     df['avg'] = df.mean(axis=1)
553     df = df.sort_values(by=['avg'], ascending=False)
554     print(df)
555     df = df.avg
556     print(df.head(25))
557     return df
558
559 def keep_columns(df, columns_keep):
560     """get a dataframe and chosen columns, and keep those while dropping
561     the rest
562
563     :param df:
564     :param columns_keep:
565     :return:
566     """
567     columns_keep.append('event')
568     column_list = list(df.columns)
569     for item in columns_keep:
570         if item in column_list:
571             column_list.remove(item)
572
573     for item in column_list:
574         print('removing column:', item)
575         df = df.drop(item, 1)
576
577     return df

```

B.5 Model Construction

```

1 import pickle
2 import os
3 import matplotlib.pyplot as plt
4 import pandas as pd
5 from sklearn.ensemble import RandomForestRegressor
6 from sklearn.metrics import r2_score
7 from sklearn.model_selection import train_test_split
8 from sklearn.neighbors import KNeighborsRegressor
9 from sklearn.naive_bayes import GaussianNB
10 from sklearn.svm import SVR
11
12
13 def load_df(filename):
14     """Loads df if filename is valid
15
16     :param filename:
17     :return:
18     """
19
20     path = r'C:\Users\New User\PycharmProjects\engg460-research-project\
Data'
21     df = pd.read_pickle(os.path.join(path, filename))
22     return df
23
24
25 def load_model(filename):
26     """Load model given filename. Must be in same directory as project.
27
28     :param filename:
29     :return:
30     """
31
32     path = r'C:\Users\New User\PycharmProjects\engg460-research-project\
Models'
33     path = os.path.join(path, filename)
34     loaded_model = pickle.load(open(path, 'rb'))
35
36     return loaded_model
37
38
39 def train(data, filename=None, algo='RF'):
40     """Train a model using the random forest regressor algorithm given a
41     dataset indexed by date.
42     Model is saved with Pickle, if filename is specified.
43     Event occurrence values MUST be under header 'event'.
44
45     :param algo:
46     :param data:
47     :param filename:
48     :return:
49     """
50     y = data.event
51     x = data.drop(['event'], axis=1)

```

```

50     print(y)
51     print(x)
52
53     x_train, x_test, y_train, y_test = train_test_split(x, y, test_size
=0.2)
54     if algo == 'KNN':
55         model = KNeighborsRegressor(n_neighbors=2)
56         model.fit(x_train, y_train)
57     elif algo == 'SVM':
58         model = SVR(gamma='scale', C=1.0, epsilon=0.2)
59         model.fit(x_train, y_train)
60     else:
61         rf = RandomForestRegressor(n_estimators=1000)
62         model = rf.fit(x_train, y_train)
63
64     predictions = model.predict(x_test)
65     print("Score:", model.score(x_test, y_test))
66     print("R-Squared:", r2_score(y_test, predictions))
67     df = pd.DataFrame({'Actual': y_test, 'Predicted': predictions})
68     df1 = df.head(25)
69     print(df1)
70     df1.plot(kind='bar', figsize=(16, 10))
71     plt.grid(which='major', linestyle='-', linewidth='0.5', color='green
')
72     plt.grid(which='minor', linestyle=':', linewidth='0.5', color='black
')
73     plt.show()
74     if filename is not None:
75         path = r'C:\Users\New User\PycharmProjects\engg460-research-
project\Models'
76         path = os.path.join(path, filename)
77         pickle.dump(model, open(path, 'wb'))
78         model = load_model(filename)
79         new_predictions = model.predict(x)
80         print("New score:", model.score(x, y))
81         print("New R-Squared:", r2_score(y, new_predictions))
82
83     return
84
85
86 def test_model(data, filename, plot=False):
87     """Test a model given dataset and model file.
88
89     :param data:
90     :param filename:
91     :param plot:
92     :return:
93     """
94     print(data)
95     y = data.event
96     x = data.drop(['event'], axis=1)
97     model = load_model(filename)

```

```

98 predictions = model.predict(x)
99 print("Score:", model.score(x, y))
100 df = pd.DataFrame({'Actual': y, 'Predicted': predictions})
101 df = df.sort_index()
102 print(df)
103 if plot:
104     ax = plt.gca()
105     df = df.reset_index()
106     df.plot(x='date', y='Actual', label='Predicted values', ax=ax)
107     df.plot(x='date', y="Predicted", label='Actual values', ax=ax)
108     plt.xlabel('Date')
109     plt.ylabel('Event Occurrence')
110     plt.legend(loc='upper right')
111     plt.show()
112
113     df.set_index('date', inplace=True)
114
115     # feature_importances = pd.DataFrame(model.feature_importances_,
116     #                                     index=x.columns,
117     #                                     columns=['importance']).
118     sort_values('importance', ascending=False)
119     # print(feature_importances)
120     return df

```

B.6 Detection Functions

```

1 import matplotlib.pyplot as plt
2 import pandas as pd
3 import pickle
4 import os
5 from datetime import timedelta, datetime
6
7 from Detector.database import get_data_between_dates, get_peril_dates
8
9
10 def create_confidence_interval(df, threshold, filename=None):
11     """Creates a confidence interval from an input dataframe.
12     Takes all predicted values under a certain threshold, and outputs
13     confidence intervals of event occurrences
14
15     :param filename:
16     :param threshold:
17     :param df:
18     :return:
19     """
20     dates = []
21     for row in df.itertuples():
22         predicted_val = row.Predicted

```



```
22     date = row.Index
23
24     if predicted_val < threshold:
25         confidence = 1 - (predicted_val/threshold)
26         date_dict = {
27             'date': date,
28             'confidence': confidence
29         }
30         dates.append(date_dict)
31     confidence_df = pd.DataFrame(dates)
32
33     confidence_df.set_index('date', inplace=True)
34     print(confidence_df)
35
36     if filename is not None:
37         path = r'C:\Users\New User\PycharmProjects\engg460-research-
38         project\Data'
39         path = os.path.join(path, filename)
40         confidence_df.to_pickle(path)
41     return confidence_df
42
43 def group_events(df):
44
45     group_date = None
46     dates = []
47     df_length = len(df.index)
48     i = 0
49     for row in df.itertuples():
50         if group_date is None:
51             group_date = row.Index
52             group_confidence = row.confidence
53         else:
54             date = row.Index
55             confidence = row.confidence
56             difference = (date - group_date).days
57             if difference <= 3:
58                 if confidence > group_confidence:
59                     group_confidence = confidence
60             else:
61                 date_dict = {
62                     'date': group_date,
63                     'confidence': group_confidence
64                 }
65                 dates.append(date_dict)
66                 group_date = row.Index
67                 group_confidence = row.confidence
68         i += 1
69     if i == df_length:
70         date_dict = {
71             'date': group_date,
72             'confidence': group_confidence
```

```

73         }
74         dates.append(date_dict)
75     filtered_df = pd.DataFrame(dates)
76     filtered_df.set_index('date', inplace=True)
77
78     return filtered_df
79
80
81 def filter_known_events(df, peril_database, peril_collection, begin_date
82 , end_date):
83     unknown_events = []
84     known_events = []
85     dates = []
86     peril_data = get_peril_dates('Tropical Cyclone', begin_date,
87 end_date, peril_database, peril_collection)
88     peril_len = len(peril_data)
89     for item in peril_data:
90         print(item)
91
92     for row in df.itertuples():
93         known_event = False
94         date = row.Index
95         i = -3
96         date_dict = {}
97
98         while i < 10:
99             date = date + timedelta(i)
100             if date in peril_data:
101                 known_event = True
102                 peril_data.remove(date)
103                 date_dict = {
104                     'date': row.Index,
105                     'confidence': row.confidence,
106                     'event': True
107                 }
108             i += 1
109         if known_event is False:
110             date_dict = {
111                 'date': row.Index,
112                 'confidence': row.confidence,
113                 'event': False
114             }
115         unknown_events.append(row)
116         known_events.append(row)
117         dates.append(date_dict)
118
119     all_events = pd.DataFrame(dates)
120     all_events.set_index('date', inplace=True)
121     print(all_events)
122     print(len(known_events), 'perils found out of', peril_len)
123     print(peril_data)

```

```

123
124 true_pos = len(known_events)
125 false_neg = peril_len-len(known_events)
126 false_pos = len(all_events) - len(known_events)
127
128 print('True Positives:', true_pos)
129 print('False Negatives:', peril_len-len(known_events))
130 print('False Positives:', len(all_events) - len(known_events))
131
132 precision = true_pos/(true_pos + false_pos)
133 recall = true_pos/(true_pos + false_neg)
134
135 if true_pos == 0:
136     f1_score = 0
137 else:
138     f1_score = 2*((precision*recall)/(precision+recall))
139 print('Precision:', precision)
140 print('Recall:', recall)
141 print('F1 Score', f1_score)
142
143 return all_events
144
145
146 def return_ids(df, article_database, article_collection,
147               minimum_confidence, proximity):
148     """
149     :param proximity:
150     :param df:
151     :param article_database:
152     :param article_collection:
153     :param minimum_confidence:
154     :return:
155     """
156     all_ids = []
157     for row in df.itertuples():
158         ids_in_range = []
159         if row.confidence > minimum_confidence:
160             begin = row.Index
161             end = row.Index + timedelta(days=proximity)
162             print('Begin is:', begin, 'and end is:', end)
163             data = get_data_between_dates(begin, end, article_database,
164 article_collection)
165             i = 0
166             for item in data:
167                 article_id = item.get('id')
168                 ids_in_range.append(article_id)
169                 i += 1
170             if ids_in_range:
171                 entry = {
172                     'begin': begin,

```

```
173         'confidence': row.confidence,
174         'ids': ids_in_range
175     }
176     all_ids.append(entry)
177
178 all_ids = pd.DataFrame(all_ids)
179
180
181 print(all_ids)
182 return all_ids
```

Bibliography

- [1] R. Crompton, “Normalising the Insurance Council of Australia Natural Disaster Event List: 1967–2011,” 2011.
- [2] C. Manning, P. Raghavan, and H. Schütze, *Introduction to information retrieval*. New York, NY, USA: Cambridge: Cambridge University Press, 2018.
- [3] S. Chipman, *The Oxford handbook of cognitive science*. Oxford University Press, 2017.
- [4] D. Roth, “Learning to resolve natural language ambiguities: A unified approach.,” pp. 806–813, 1998.
- [5] B. Broda, P. Kędzia, M. Marcińczuk, A. Radziszewski, R. Ramocki, and A. Wardyński, “Fextor: A Feature Extraction Framework for Natural Language Processing: A Case Study in Word Sense Disambiguation, Relation Recognition and Anaphora Resolution,” *Computational Linguistics*, pp. 41–62, 2013.
- [6] R. Krishnamurthy, Y. Li, S. Raghavan, F. Reiss, S. Vaithyanathan, and H. Zhu, “SystemT: A System for Declarative Information Extraction,” 2009.
- [7] P. Nadkarni, L. Ohno-Machado, and W. Chapman, “Natural language processing: an introduction,” *Journal of the American Medical Informatics Association*, pp. 544–551, 2011.
- [8] J. Webster, “TOKENIZATION AS THE INITIAL PHASE IN NLP,” *Proceedings of the fifteenth International Conference on Computational Linguistics*, 1992.
- [9] D. Lewis and K. Jones, “Natural language processing for information retrieval,” *Communications of the ACM*, p. 92–101, 1996.
- [10] K. Kimmo, “To stem or lemmatize a highly inflectional language in a probabilistic IR environment?,” *Journal of Documentation*, pp. 476–496, 1992.
- [11] H. Liu, T. Christiansen, Baumgartner, W.A., and K. Verspoor, “BioLemmatizer: a lemmatization tool for morphological processing of biomedical text,” *Journal of Biomedical Semantics*, p. 3, 2012.

- [12] A. Zelaia, I. Naki Alegria, O. Arregi, and B. Sierra, “Analyzing the Effect of Dimensionality Reduction in Document Categorization for Basque,” *Archives of Control Sciences*, 2005.
- [13] V. Baradad and A. Mugabushaka, “Corpus Specific Stop Words to Improve the Textual Analysis in Scientometrics,” 2015.
- [14] A. Tellez Valero, M. Montes y Gomez, and L. Villasenor Pineda, “Using Machine Learning for Extracting Information from Natural Disaster News Reports,” *Computacion y Sistemas*, pp. 33–44, 2009.
- [15] N. Ireson, F. Ciravega, and M. E. Califf, “Evaluating Machine Learning for Information Extraction,” 2005.
- [16] X. Zhou and C. Xu, “Tracing the Spatial-Temporal Evolution of Events Based on Social Media Data,” *International Journal of Geo-Information*, pp. 1–15, 2017.
- [17] H. Sayyadi, A. Sahraei, and H. Abolhassani, “Event Detection from News Articles,” *Advances in Computer Science and Engineering*, pp. 981–984, 2019.
- [18] T. Lane and C. Brodley, “An application of machine learning to anomaly detection,” pp. 1–13, 1997.
- [19] T. G. Dietterich, “Machine learning for sequential data: A review,” in *Structural, Syntactic, and Statistical Pattern Recognition* (T. Caelli, A. Amin, R. P. W. Duin, D. de Ridder, and M. Kamel, eds.), (Berlin, Heidelberg), pp. 15–30, Springer Berlin Heidelberg, 2002.
- [20] Nla.gov.au., “Api overview | help centre,” 2019.
- [21] Nla.gov.au., “Optical character recognition (ocr) on newspapers,” 2019.
- [22] Nla.gov.au., “Api version 2 technical guide | help centre,” 2019.
- [23] “Beautiful soup documentation — beautiful soup 4.4.0 documentation,” 2019.
- [24] Nltk.org., “2. accessing text corpora and lexical resources,” 2019.
- [25] Pydata.org., “Python data analysis library — pandas: Python data analysis library,” 2019.
- [26] Scikit-learn.org., “sklearn.featureextraction.text.tfidfvectorizer — scikit-learn 0.20.3 documentation.r,” 2019.
- [27] Scikit-learn.org., “sklearn.ensemble.randomforestregressor — scikit-learn 0.20.3 documentation.,” 2019.
- [28] Python.org., “Pep 287 – restructuredtext docstring format.,” 2019.
- [29] Herokuapp.com, “Trove api console,” 2019.