

# Distributed Email Processing Pipeline

A dissertation submitted in partial fulfilment of  
the requirements for the degree of  
BACHELOR OF *ENGINEERING* in Computer Science  
in  
The Queen's University of Belfast  
by  
Cathal Mullan

Friday, 24 April 2020

**SCHOOL OF ELECTRONICS, ELECTRICAL ENGINEERING and COMPUTER  
SCIENCE**

**CSC3002 – COMPUTER SCIENCE PROJECT**

**Dissertation Cover Sheet**

A signed and completed cover sheet must accompany the submission of the Software Engineering dissertation submitted for assessment.

Work submitted without a cover sheet will **NOT** be marked.

**Student Name:** Cathal Mullan

**Student Number:** 40180175

**Project Title:** Distributed Email Processing Pipeline

**Supervisor:** Dr Barry Devereux

**Declaration of Academic Integrity**

Before submitting your dissertation please check that the submission:

1. Has a full bibliography attached laid out according to the guidelines specified in the Student Project Handbook
2. Contains full acknowledgement of all secondary sources used (paper-based and electronic)
3. Does not exceed the specified page limit
4. Is clearly presented and proof-read
5. Is submitted on, or before, the specified or agreed due date. Late submissions will only be accepted in exceptional circumstances or where a deferment has been granted in advance.  
**By submitting your dissertation you declare that you have completed the tutorial on plagiarism at <http://www.qub.ac.uk/cite2write/introduction5.html> and are aware that it is an academic offence to plagiarise. You declare that the submission is your own original work. No part of it has been submitted for any other assignment and you have acknowledged all written and electronic sources used.**
6. If selected as an exemplar, I agree to allow my dissertation to be used as a sample for future students. (Please delete this if you do not agree.)

*Student's signature*

Cathal Mullan

*Date of submission*

24/04/20

## Table of Contents

<b>1</b>	<b><i>Acknowledgements</i></b> .....	<b>5</b>
<b>2</b>	<b><i>Abstract</i></b> .....	<b>5</b>
<b>3</b>	<b><i>Introduction and Problem Area</i></b> .....	<b>5</b>
<b>4</b>	<b><i>Solution Description and System Requirements</i></b> .....	<b>6</b>
4.1	Email Source .....	6
4.2	Intermediate Storage .....	7
4.3	Email Storage Consumer .....	7
4.4	Machine Learning Pre-Processing.....	7
4.5	Machine Learning Jobs .....	8
4.5.1	Topic Modelling .....	8
4.5.2	Summarization.....	8
4.6	Model Deployment .....	8
<b>5</b>	<b><i>Design</i></b> .....	<b>9</b>
<b>6</b>	<b><i>Implementation</i></b> .....	<b>10</b>
6.1	Data Gathering.....	10
6.1.1	Email Crawler.....	10
6.1.2	Email Generator.....	12
6.1.3	Enron Dataset .....	12
6.2	Intermediate Storage .....	12
6.3	Email Event Consumer .....	13
6.3.1	Stream Processing .....	13
6.3.2	Email Anonymization.....	16
6.4	Machine Learning Pre-Processing.....	17
6.5	Machine Learning Jobs .....	17
6.5.1	Topic Modelling .....	17
6.5.2	Summarization.....	18
6.6	Distributed Training .....	18
6.7	Model Serving.....	19

6.8	Pipeline Deployment.....	20
6.9	Monitoring and Alerts .....	21
<b>7</b>	<b><i>Testing, Validation and Code Quality.....</i></b>	<b>22</b>
7.1	Testing .....	22
7.1.1	Email Crawler.....	22
7.1.2	Email Event Consumer .....	22
7.1.3	Machine Learning Jobs .....	23
7.2	Continuous Integration .....	24
<b>8</b>	<b><i>System Evaluation and Experimental Results.....</i></b>	<b>25</b>
8.1	Email Crawler.....	25
8.2	Kafka Storage.....	25
8.3	Spark Streaming and Batch Processing .....	26
8.4	Email Anonymization .....	26
8.5	Topic Modelling .....	27
8.6	Summarization.....	28
8.7	Model Predictions .....	29
8.8	Deployment .....	31
<b>9</b>	<b><i>Closing Thoughts .....</i></b>	<b>31</b>
<b>10</b>	<b><i>References.....</i></b>	<b>32</b>

# **1 Acknowledgements**

I would like to acknowledge my supervisor Dr Barry Devereux for his advice and guidance surrounding the machine learning techniques used within this project, as his expertise in this topic significantly clarified the direction of this project.

I would also like to thank Proofpoint for their financial contribution to the deployed pipeline, along with architectural guidance provided for this distributed solution.

# **2 Abstract**

Over 250 billion emails are sent every day (The Radicati Group, 2015). There is great potential within this data, and in order to extract meaning from even a fraction of these emails, a robust, scalable solution is necessary. Working in conjunction with Proofpoint, a pipeline has been created which attempts to tackle this task, while showcasing the usefulness of such a pipeline in tackling a real-world problem.

Email triaging is a task that many professionals find themselves slugging through daily. Through the use of trained machine learning models, this task can be aided greatly. In order to train these models on a continuous stream of data, along with a robust, resilient pipeline will be required, with a specific focus on automation.

# **3 Introduction and Problem Area**

Email management can take up a significant amount of times for individuals in the workplace, who often find themselves bombarded with them constantly. The average worker can find themselves spending 28% of their work week dealing with this task (Michael Chui, 2012). Attempting to identify which emails are worth opening and reading can be cumbersome and slow. With an improved system, there could potentially be a rise in general productivity of an individual.

This project describes several machine learning models which can be used to aid the management of emails and improve general efficiency of a task many of us are required to perform numerous times each day, effectively assisting email triaging.

The first model allows for automated filtering of emails into one of a number of predetermined categories. Such a task would usually be tackled by the end user through means such as filtering manually based on email headers, or alternately using services such as Gmail's filtering into a handful of non-specific categories (Google, 2020). Through the use of this model, such filtering will be enhanced by the addition of more granular categories.

The second model will be a summarization model, which will predict the subject of an email based on the contents of the body. The vision here is to replace the standard view of an email with a generated summarization, as opposed to the standard truncated email body usually shown by email clients. Such a model will allow the user to gain a greater understand of the contents of an email without having to read it completely.

The data used will not be a traditional dataset, but rather a continuous stream of email files, in order to mimic a real-world application of data gathering. As such, the pipeline created must be scalable depending of the size of data to be processed per day, and automated to train the models daily, utilizing both streaming and batch processing to do so.

## **4 Solution Description and System Requirements**

### **4.1 Email Source**

As mentioned previously, there is an expectation to create a source of email files that can be gathered over time. While there are a number of popular email datasets, such as the Enron dataset consisting of around half a million emails, in order to reach the scale of emails being aimed for, an additional data source is required.

The choice settled on was the website MARC archiving site, an email archive consisting of mailing lists from a number of open source projects. MARC contains over seventy million unique emails, which will help meet the expected dataset scale (Leininger, 2014).

Due to this being a website, in order to gather these mails, a website crawler would be required. This crawler should be able to parse all the varieties of pages within the MARC site and produce emails to be consumed elsewhere.

Such a crawler may not produce enough emails to validate the scalability of the pipeline. As such, a generator will be required to produce a large scale of emails at once, allowing a mock scenario for usage in testing.

## **4.2 Intermediate Storage**

As emails are being gathered, there needs to be a temporary storage before emails are processed and stored externally. While there is a possibility that these emails discovered by the email crawler could be processed immediately, in order to manage potential surges in produced emails, an intermediate storage is required. Such a system is popular in event driven architecture and fits this style of processing nicely.

This storage will contain compressed text blobs of the raw email files and will store them for a minimum of twenty-four hours before being purging. This should allow enough time to recover from issues that may occur further down the pipeline. It must be able to scale linearly not only in terms of writing into storage but also consuming from it as well.

## **4.3 Email Storage Consumer**

In order to process the emails within the temporary storage, a continuous streaming processing system is required. This system again must be horizontally scalable to allow processing of any number of email events. It must be resilient when processing invalid data, and selective regarding the quality of emails.

This will act as the initial processing stage for the emails and should be minimal in its transformation of emails, instead focusing on ensuring the correctness of the emails, parsing them into easily processable structures. These processed emails should then be saved in a permanent storage system given they pass the validation process.

## **4.4 Machine Learning Pre-Processing**

Each of the subsequent machine learning jobs will require a separate format of data for the training. As such, a number of batch jobs will be required to transform the data, and act as the precursor to the machine learning jobs.

These jobs should automatically read the data stored from the previous day of streaming and produce specific datasets to be used for training.

Again, these jobs should act as an additional layer of validation depending of the downstream job. Based on the specific job, this may require selecting emails which have sufficient body length or are the head of the conversation (e.g. without a reply tag in the subject).

## **4.5 Machine Learning Jobs**

### **4.5.1 Topic Modelling**

This model will allow for the sorting of email files into one of a number of predetermined topics. The underlying natural language processing technique that will be used is latent Dirichlet allocation (LDA) (David M. Blei, 2003). The starting point for the code will be an example script provided by Google using their TensorFlow library (The TensorFlow Probability Authors, 2018).

### **4.5.2 Summarization**

This model will produce a summarization of an email utilizing the email body and subject. The technique used here will be sequence-to-sequence (Seq2Seq) (Ilya Sutskever, 2014), and use the email body and the initial text, and the subject as the targeted string. Again, a starting point for the code was chosen, being an example repository found on GitHub (Chen, 2018) which extends a Keras provided tutorial (Marmiesse, 2019).

## **4.6 Model Deployment**

In order to benefit from these models, they must be served over an API. The payload of such an API should be a raw email file, and in response the predictions from the models should be returned. This API should be simple to use and return a parsable JSON response to ease the potential integration of this API with standard email processing systems, similarly to how virus scanners are integrated.



## 5 Design

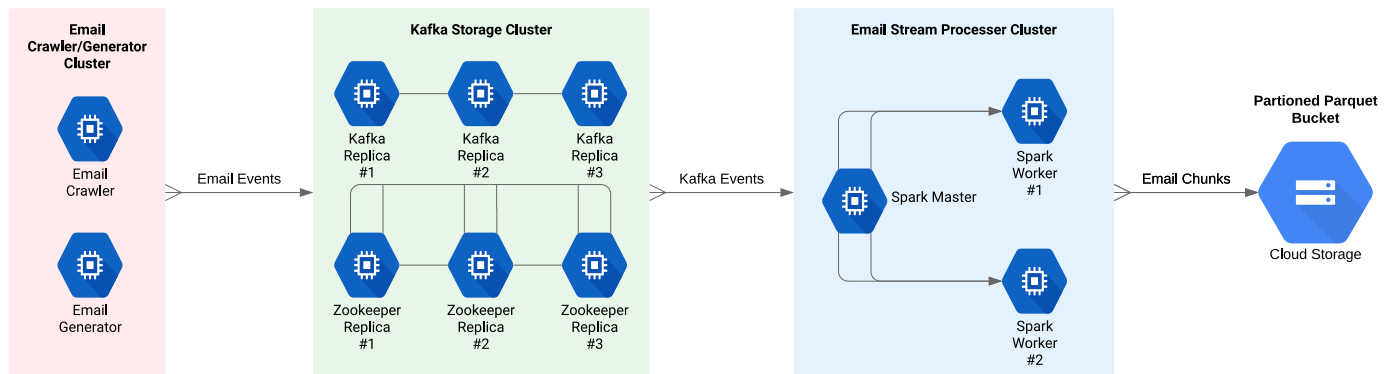


Figure 1: Data Gathering

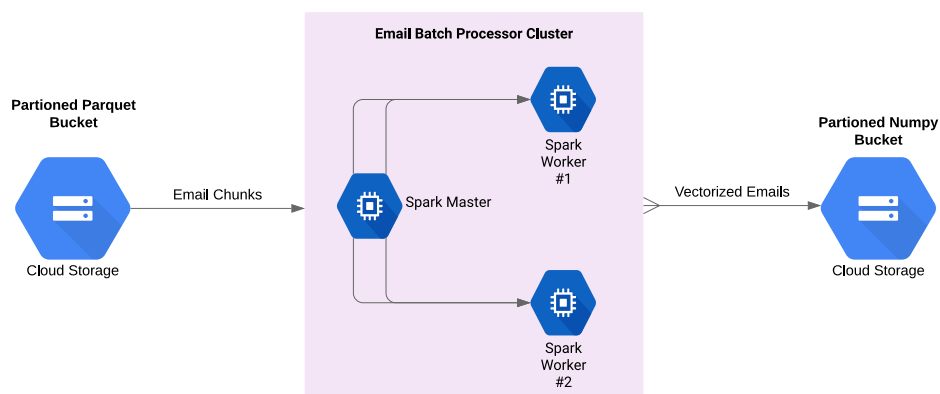


Figure 2: Machine Learning Pre-processing

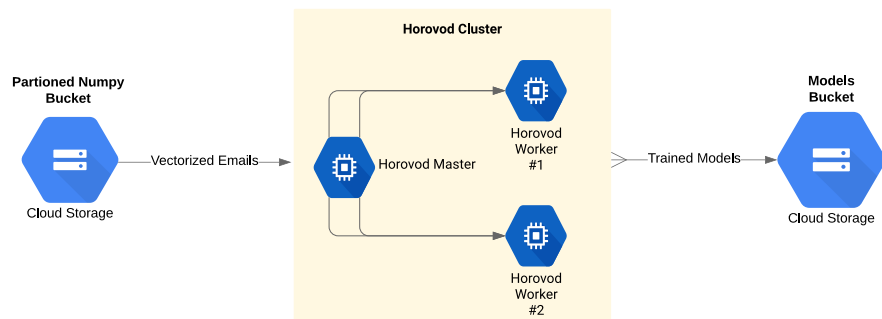
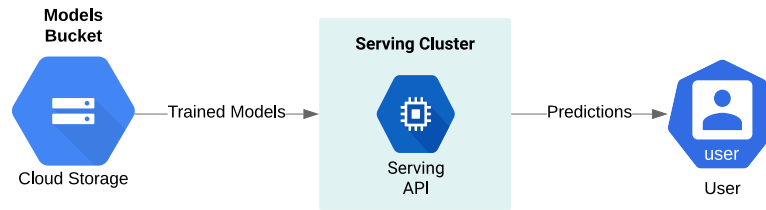


Figure 3: Machine Learning Jobs



*Figure 4: Model Serving*

## 6 Implementation

### 6.1 Data Gathering

#### 6.1.1 Email Crawler

Perhaps the simplest component of this pipeline, the email crawler is a web crawling bot implemented in Python. Utilizing the Scrapy framework (Scrapy developers, 2020), this bot allows for the gathering of emails from a public archiving site. The site chosen was the MARC mailing archives, containing over seventy million emails from mailing lists from a number of open source projects.

In terms of the process of scraping, this site is rather minimal and does not ship with any JavaScript (JS) or blocking cascading style sheets (CSS), providing an optimal environment for scraping. There are only a handful of page types within this site, allowing for the crawler to be minimal in terms of implementation.

There are a number of caveats involved with this bot. As mentioned earlier, this site aggregates emails from open source projects, which results in data which is highly technical in contents. As such, there is a concern that this will not reflect the typical email style expected to be encountered in casual conversations. This could cause issues both in the quality of the final model and with the email parsing used. Also, these technical emails often include code blocks. Any words within these code blocks may be interpreted as text in a sentence, which could incorrectly lead to jargon data being swallowed by the models. Alongside this issue, there is a deliberate mangling of the emails implemented by MARC in order to prevent malicious crawlers from gathering email addresses for spam purposes. Such mangling needs to be circumvented.

[[prev in list](#)] [[next in list](#)] [[prev in thread](#)] [[next in thread](#)]

List: [trustedbsd-audit](#)  
Subject: [OpenBSM moving to GitHub](#)  
From: [Christian Brueffer <brueffer \(\) FreeBSD ! org>](#)  
Date: [2015-07-01 8:43:47](#)  
Message-ID: [5593A843.8030800 \(\) FreeBSD ! org](#)  
[Download RAW [message](#) or [body](#)]

We're pleased to announce the move of the OpenBSM source code repository from the FreeBSD Perforce server to Github. After a period of dormancy, we hope this will make the code more accessible and stimulate outside contributions.

Since the converter (git-p4) could not export the release labels from Perforce correctly, they were added to the git repository by hand. Thus, don't be alarmed by the recent tagging dates.

The repository can now be found at <https://github.com/openbsm/openbsm>

Christian Brueffer and Robert Watson on behalf of the OpenBSM project.

---

trustedbsd-audit@FreeBSD.org mailing list  
<http://lists.freebsd.org/mailman/listinfo/trustedbsd-audit>  
To unsubscribe, send any mail to "trustedbsd-audit-unsubscribe@FreeBSD.org"  
[[prev in list](#)] [[next in list](#)] [[prev in thread](#)] [[next in thread](#)]

*Figure 5: Example email from marc.info*

Another issue that faces this bot is the Robot policy of the MARC site. These policies typically outline the expected behaviour of bots that interact with the site, often including limits on requests that can be made per second or minute. This is implemented in a text file found at the top directory of the site and is considered good practice to adhere to. While these can be bypassed, it is a sign of good will to adhere to these rules. The issue here is that their policy disallows all forms of bots outright. However, by reading further on the MARC site, we can see that these policies are simply in place to prevent malicious bots, and that there is an acceptance of bots which adhere to request limiting of 30 requests per minute. This effectivity limits the amount of emails which can be gathered from using this crawler to around 40 thousand per day.

The crawling performed here resulted in a generated event containing the email as a payload, with no additional pre-processing. The concept at this stage was to allow for a singular interface to handle email processing, instead of requiring each data source to roll its own processing. As such, the payload was a highly compressed email file, utilizing Google's Snappy compression (Google, 2011). Snappy was chosen over the more typical Zstandard compression (Facebook, 2015), as while it resulted in larger files, the processing power required was minimal. Additional metadata was added to provide reference to which mailing lists an email originated from, and the date of gathering. This processing was based off an example found on GitHub, which crawled the same MARC site in order to generate an XML output (Alcaras, 2018).

Regarding the deployment of this crawler, we utilized long-living, small instances to allow for the bot to run continuously without disturbance. This allowed the crawler to maintain a cache of emails already parsed locally, preventing accidentally processing of the same emails more than once. In order to scale this deployment, an external cache would be required. As such, only a single instance was used. This aligns with the caveat in terms of request per minute that had to be adhered to, as additional bots would be simply be a waste since a single bot was already hitting the request limits. If such a limit was not in place, a Redis caching layer would have been implemented to allow for a number of crawlers to run concurrently.

### **6.1.2 Email Generator**

Due to the issues mentioned above regarding the rate of new emails that can be gathered using the crawler (around 30 per minute), in order to gauge the scalability side of this pipeline, a solution was needed to produce emails at a greater scale. The solution settled on was using an example email to simply produce dummy events at a set rate per second. This allowed me to gradually or dramatically increase the rate of email production, allowing scalability testing.

### **6.1.3 Enron Dataset**

In order to develop models locally, the Enron dataset of emails (Cohen, 2015) was utilized. This famous dataset contains around half a million emails, often business related, which were released as a result of Enron's collapse in 2001. Having such a dataset available helped with validating that a model showed potential before being implemented within the pipeline for training over time.

## **6.2 Intermediate Storage**

Aligning with standard stream processing, there is a requirement for a distributed storage solution to provide temporary storage of events before they are processed. This can be used as a buffer to manage scaling issues that can occur with surges in events. Such a storage can also prevent crises that may occur if processing was to halt for a period of time.

Apache Kafka (Apache Software Foundation, 2011) is one such implementation of a distributed storage solution. Using Kafka, events are received from ‘producers’, and then served to ‘consumers’, which is often described as the publish subscribe (Pub/Sub) style of storage streaming. Producers generate events that refer to a specific ‘topic’, which acts as an identifier or tag to the data type. In order to allow for scaling of this storage solution, data is partitioned across a number of nodes. While unimportant to this pipeline, Kafka runs on top of an additional framework, Apache Zookeeper (Apache Software Foundation, 2008), to allow for the syncing of data between nodes. As such, for each Kafka node created, a subsequent Zookeeper node is a requirement.

Through this partitioning, data can be extracted from nodes in a parallelized fashion. When reading from a Kafka topic, an ‘offset’ is used to extract events between a certain time frame. This can also be utilized to continuously stream data as it is gathered, while maintaining a checkpoint offset to begin processing again at a later date in the event of an outage. Once an event had been processed, it can be tagged to prevent reprocessing in the future.

As mentioned above, Kafka is a dynamic, temporary storage. Events can be stored for a specific period of time before being deleted. I have opted to choose a retention period of 24 hours. This allows me enough time to deal with outages effectively. Due to the nature of this processing pipeline, there is not a requirement for every single event to be processed like with other application of Kafka, such as banking event processing. As such, there is no concern with missing events beyond the decrease in efficiency of the pipeline that would occur as a result.

## **6.3 Email Event Consumer**

### **6.3.1 Stream Processing**

As data is being stored in the Kafka cluster, we need a system to read this data and provide further processing in a distributed fashion. There are a number of frameworks which allow for reading from Kafka, but specifically we were looking for one that can perform both batch and stream processing. The two contenders in this space are Apache Spark (Zaharia, 12) and Apache Flink (Apache Software Foundation, 2011). Spark ultimately was chosen, as although it functions on a batch-processing first approach, its interconnectivity with external resources was much more sophisticated than Flink, mainly stemming from its longevity in the big data processing space.

Spark itself is implemented using Scala, but due to the usage of Python already in this project, PySpark, the Spark Python binding library, was chosen. The main caveat with this decision is that in order to run Python directly on the dataset, there is a cost involved with serializing data between the Java Virtual Machine (JVM) and the Python runtime. However, the increase in productivity gained from using a more familiar language justified this decision. It is worth mentioning also due to the ease of scalability involved with using this library and pipeline, the performance hit can be somewhat disregarded.

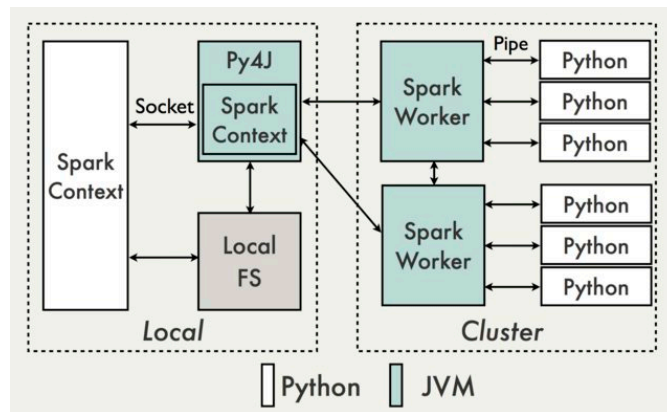


Figure 6: Python to JVM data translation

This stage of processing performs the standard email parsing and validation, specifically that of MIME parsing. Emails are pulled from Kafka, partitioned by date, decompressed, then serialized into an 'EmailMessage' class, functionality which is provided by the Python standard library. This parsing is run using the strictest policy, resulting in only perfect emails without defects being accepted. On reflection, this parsing is overly harsh, often resulting in emails with minimal issues being rejected. This could affect the models down the line, as a great deal of emails in the wild contain one or two issues, which the model will not have encountered before. Thankfully, most of these issues reside in the email headers, and not the subject or body which is relevant to the training and predictions performed.

Once we have verified an email is valid, the next stage extracts specific components of the email into a more efficient structure. The core of the processing involves the subject and body components. The subject is cleaned by removing prefixes such as 'RE:' and 'FWD:', removing irrelevant tokens. There is no specific requirement that an email needs to have a subject beyond an empty string, so it is an optional field.

The body undergoes much more complex processing. Email bodies can consist of a number of components, and as such, we attempt to extract the primary text section. We do not want the machine learning pipeline to process additional text which is not relevant to the specific email being processed, as it would slow down processing and likely lead to multiple replies being processed at once. If there is no plaintext section, we will fall back and extract the primary HTML component, and then leverage HTML parsing to extract the text from within this code. Once this section has been extracted, we next remove inline messages, which often exist as a result of replying to an email. This concept of email responses is not strictly standardized and differs greatly between email providers and clients. An attempt is made to identify markers inserted by Microsoft's Outlook and Google's Gmail specifically, but other variants may slip through this stage. As such, MailGun's Talon (MailGun, 2019) library was utilized, which uses machine learning to identify core text from that of inline text. This library also attempts to remove signatures from the body, which provides additional benefits of removing likely irrelevant text.

```
MIME-Version: 1.0
Content-Type: multipart/mixed; boundary=frontier

This is a message with multiple parts in MIME format.
--frontier
Content-Type: text/plain

This is the body of the message.
--frontier
Content-Type: application/octet-stream
Content-Transfer-Encoding: base64

PGh0bWw+CiAgPGhlYWQ+CiAgPC9oZWFKPgogIDxib2R5PgogICAgPHA+VGhpcyBpcyB0aGUg
Ym9keSBvZiB0aGUgbWVzc2FnZS48L3A+CiAgPC9ib2R5Pgo8L2h0bWw+Cg==
--frontier--
```

*Figure 7: A multipart MIME email body.*

Once all components have been extracted, there is a final check before accepting the email contents are valid, which specifically ensures the email body is not below a threshold figure of below 200 or above 5000 characters long. Assuming this is fine, the email will be added to a data frame structure to be stored externally.

The data format chosen to store these processed emails in the Apache Parquet (Twitter, Cloudera, 2013) format. It functions similarly to a CSV file; in that it is a column-oriented storage format, but with strict data types included alongside the stored data. However, it also includes benefits when extracting data from an external source. The data can be queried from within a storage solution, effectively allowing subsets of data to be extracted efficiently from storage. This is possible due to splitting data in chunks while storing metadata which can be used to reference which chunks contain the relevant data requested for. Alongside this benefit, it is much more efficient in terms of compression, resulting in smaller dataset sizes. Again, Snappy was used for compression. In summary, Parquet provides the benefits offered from structured data sources such as databases, combined with the ease of use provided by standard files.

Once a chunk of data has been gathered, it is streamed into a distributed object store bucket. In the production environment, the Google Cloud Platform (GCP) was selected as the cloud provider of choice, so here we used Google Cloud Storage (GCS). This is equivalent to the more popular S3 storage offered by Amazon. As the data is structured by date, the sub-folder within the bucket referenced the date of processing.

### **6.3.2 Email Anonymization**

Initially during the planning stage of this project, the concept of processing real-world data provided by Proofpoint was discussed. In order to comply with the security expectations of customers when processing their data, a layer of anonymization was created that could optionally be run before storing emails externally. This layer aims to remove personally identifiable data from the emails.

The first stage of this anonymization involved tagging all entities within the text. This was achieved through the usage of the spaCy library (Explosion AI, 2016), a natural language processing library which provides a number of pre-trained models for content tagging. Once tagged, the Faker library was used to generate realistic replacements, such as generated email addresses and phone numbers within the email body.



The email headers were processed separately. In order to maintain the ability to trace emails by both the user and domain components of a given address, they were separately hashed. By doing so, the identity of the individuals is kept secret, while also maintaining the relationship and format of the data.

## **6.4 Machine Learning Pre-Processing**

In order to optimize the resource used during the machine learning jobs, there is a requirement to pre-process data using batch processing before starting said jobs. Such processing is usually quite CPU intensive, compared to the GPU intensive machine learning jobs. As the cost of the GPUs is on average more expensive, we want to optimize the time spent once requesting these resources, hence these pre-processing jobs. Thankfully, Apache Spark is also capable of functioning on a schedule. This is where we first see the automation in terms of batch processing, wherein these jobs expect to find the existing Parquet data from the previous day. However, this job can also be run against an arbitrary partition of data manually.

The core processing which occurs at this stage is lemmatizing and flattening on the dataset. This involves removing unwanted characters in text, such as stop-words (e.g. ‘and’, ‘the’, ‘is’) and converting words to their base form. As the machine learning jobs require vectorized data as the dataset input, this was also performed at this stage. Using a static dictionary, the strings within the dataset were replaced with integers. This was achieved by using the Scikit-learn library, which automated this vectorization. Once all partitions of the data have been processed in a distributed fashion, there needs to be a singular reduce (or gather) performed in order to store the dataset as a single file. The data at this stage is separated into test and train sections, before being stored in a GCS bucket.

## **6.5 Machine Learning Jobs**

### **6.5.1 Topic Modelling**

In an attempt to automate the categorization of emails into a number of topics, a topic modelling machine learning model was implemented. The statistical model settled on is latent Dirichlet allocation (LDA), basing my code on an example found in Google’s TensorFlow library.

LDA attempts to identify a pre-determined number of topics across a corpus of documents. As such, each topic is defined as a distribution of words. We begin by initialising the set number of topics across the entire dataset of documents, assigning each word a topic. An individual word within a chosen document is chosen to have an invalid topic. We assign a probability to this word and topic based on the topics included within the chosen document, and the number of times the chosen word has been assigned the same topic across the entire distribution of documents.

### **6.5.2 Summarization**

The summarization model works on the assumption that an email's subject becomes less and less relevant the further away from the root message, as replies are shared. As such, a model was conceptualized to attempt to generate more relevant subjects than those automatically added by email clients. The model is opted to use utilized sequence-to-sequence (Seq2Seq) neural networks, based on a number of examples found using the Keras library, utilizing TensorFlow as the backend.

The model begins with identifying the input sequence as a given email's body, and the target sequence being the subject. An encoder/decoder long short-term memory (LSTM) neural network pairing is used to attempt and force the input sequence into the target sequence, utilizing a training process called “teaching forcing”.

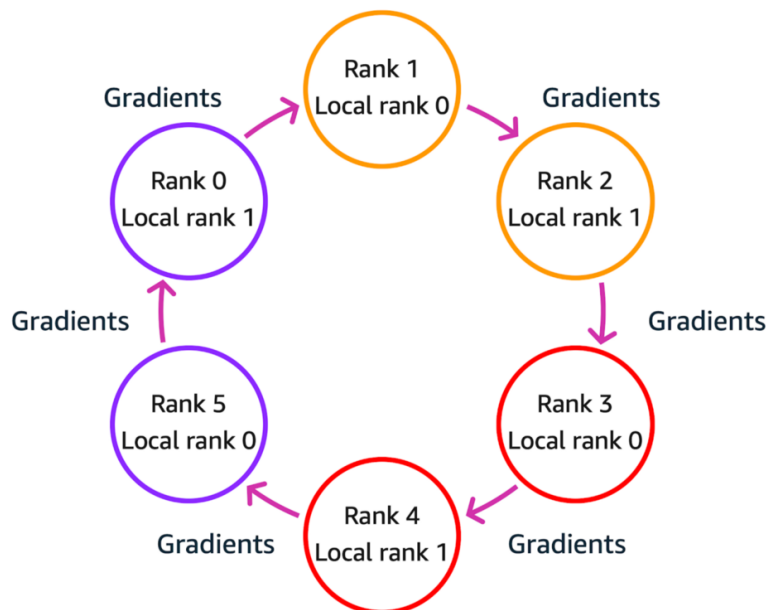
## **6.6 Distributed Training**

Machine learning as a task does not lend itself well to distribution. It is usually a highly iterative process and does not fit within the standard ‘map reduce’ style of processing, where a task can be split into an arbitrary number of partitions, processed separately then gathered as one. In order to train at scale, a solution was conceptualized by Baidu and developed by Uber.

The algorithm developed is named ‘all reduce’ (Baidu, 2017). A subset of data is distributed across a number of nodes, then the required gradients are calculated for this data. Before adjusting any weights, there is communication between each node of processing, wherein the results of the calculations are shared, often utilizing a ‘ring’ style of networking to pass data to an adjacent node to prevent saturating the messaging interface at use. This style of processing utilizes the message passing interface (MPI) to allow for this interconnectivity.

Uber created the library ‘Horovod’ (Alexander Sergeev, 2018), which implements the ring all reduce algorithm as a Python package. Horovod is compatible with all major deep learning libraries, including TensorFlow, PyTorch and Keras. While individual libraries have begun implementing all reduce processing internally, Horovod provides compatibility with all of these libraries through a single interface.

In order to convert a given machine learning job to become Horovod compatible, a small number of changes need to be made. Saving of models externally must be limited to the primary elected node (simply node number one), the rate of learning must be dynamically multiplied by the number of nodes being processed on, and the optimizer used must be broadcasted across all nodes.



*Figure 8: Illustration of ring all reduce passing gradients between a number of nodes*

## 6.7 Model Serving

Once a model has been trained, it can then be served as an API. There are alternative experimental ways to serve models, but these were found to be impractical, such as serving models through JS. The primary issue here is that only TensorFlow models are capable of such serving options, which would limit the pipelines usability across alternative libraries. The API designed is rather basic but accepts a raw email file within the request body and returns the predicted results from the chosen model. Each model is contained on its own endpoint.

## 6.8 Pipeline Deployment

In order to demonstrate the feasibility of this pipeline in a production environment, a cloud provider was needed to be selected. Google Cloud Platform was chosen due to its strong integration with technologies such as Spark and TensorFlow. However, the technologies used in this pipeline are multi-cloud capable, allowing the potential for this pipeline to be easily deployed on Amazon Web Services or Microsoft Azure if desired.

An important attribute of this pipeline is reproducibility. In order to allow this pipelines stack to be created and destroyed at will, Terraform was used. Terraform is a tool which allows for defining infrastructure requirements as declarative code. Terraform is written using a unique language, named HashiCorp Lang after the developers of Terraform, HashiCorp. Effectively, it acts as an interface with the APIs of a number of cloud providers, through the use of code.

```
resource "test_cluster" "test-cluster" {  
  name      = "streaming-cluster"  
  project   = var.project_id  
  location  = var.project_zone  
  network   = google_compute_network.test-network.self_link  
  
  remove_default_node_pool = true  
  initial_node_count       = 1  
  
  depends_on = [  
    | google_compute_network.test-network  
  ]  
}
```

*Figure 9: Snippet of Terraform code to create a test cluster of nodes.*

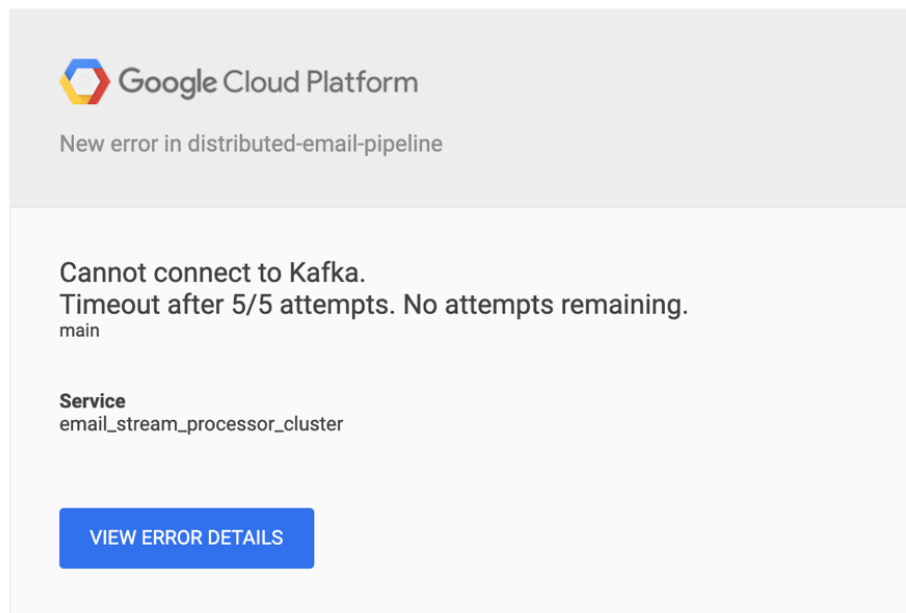
To orchestrate the deployment, there was a clear and obvious tool to choose. As Docker was used to containerize the development environment used locally, Kubernetes (Google, 2014) (K8s) would allow a similar deployment mechanism. K8s is a container orchestration system, which uses YAML templating files to automate the deployment and management of services, primarily being used in the Cloud. K8s has become the industry standard for scalable deployments. It provides a number of abstractions when working with hardware, which ease the development process greatly.

In an effort to focus on simplify, the components of the pipeline were split into separate Kubernetes clusters with single node pools. GCP provides K8s clusters free of charge, taking care of the complex task of hardware management. Clusters are essentially a group of machines (or nodes) which can be scaled horizontally, automatically, based on metrics such as CPU usage. While a cluster can contain any number of different machine types, by limiting them to a single type per cluster, it became simple to understand where a given service was running, and what hardware it was using. The caveats to this are that networking becomes a more difficult task, as K8s provides a toolkit for intra-cluster connectivity, but not connectivity between clusters.

The K8s command line tool (CLI), Kubectl, allows for deployments to be scripted easily. Once the required YAML config to deploy the services had been written, simple Bash scripts were used to automate the order of deployments. The order of deployment was decided based on the natural connectivity of resources, such as the email crawler requiring Kafka to function.

## **6.9 Monitoring and Alerts**

Due to the automated nature of the pipeline, it is paramount that the system alerts in the case of an error. Such functionality is natively built into GCP, specifically with its concept of status and pod lifecycle. A K8s pod can automatically restart itself in the event of a failure or error, which can mitigate circumstantial failures, but in the event an error cannot be recovered from, an email will be sent to the project owner. This is especially important due to the storage capabilities of Kafka being temporary. Any issues encountered during the email gathering stage must be dealt with within the Kafka retention time period, or else data will be lost.



*Figure 10: An error alert from GCP.*

## 7 Testing, Validation and Code Quality

### 7.1 Testing

#### 7.1.1 Email Crawler

To test the crawler, a local copy of a number of pages from the MARC site were stored locally to validate the parsing. This allowed for simple, minimal tests to be created, without the need of any external connectivity for testing. The expectation of these tests is to verify that the correct links would be gathered from a given web page and to ensure that all potential emails were gathered. There is no testing performed on the event generation beyond the creation of an event, as this was verified manually. Depending on opinions around whether the filesystem should be used for unit tests, these tests fall somewhere between a unit and an integration test.

#### 7.1.2 Email Event Consumer

Testing here once again involved the usage of pre-downloaded files. As this process involves MIME parsing, it was clear that the pipeline would need to be validated that valid emails are parsed as expected, while also correctly dealing with invalid emails. As such, a number of public email parsing repositories were searched for emails which caused them issues, and these faulty emails were fed through the pipeline. This turned out to be invaluable, as there was a number of problems discovered by using this method.

```

===== test session starts =====
platform darwin -- Python 3.6.9, pytest-5.1.0, py-1.8.1, pluggy-0.13.1
rootdir: /Users/cmullan/workspace/email_stream_processor, inifile: setup.cfg, testpaths: tests/
plugins: cov-2.7.1
collected 613 items

tests/email_stream_processor_tests/helpers/input/input eml_test.py ..... [ 4%]
..... [ 19%]
..... [ 24%]
tests/email_stream_processor_tests/helpers/input/input parquet_test.py . [ 24%]
tests/email_stream_processor_tests/helpers/output/output eml_test.py . [ 24%]
tests/email_stream_processor_tests/helpers/output/output parquet_test.py . [ 24%]
tests/email_stream_processor_tests/helpers/validation/address_validation_test.py ..... [ 25%]
tests/email_stream_processor_tests/helpers/validation/text_validation_test.py . [ 25%]
tests/email_stream_processor_tests/parsing/message_body_extraction_test.py ..... [ 28%]
..... [ 44%]
..... [ 47%]
tests/email_stream_processor_tests/parsing/message_contents_extraction_test.py ..... [ 49%]
..... [ 65%]
..... [ 80%]
..... [ 95%]
.. [ 96%]
tests/email_stream_processor_tests/parsing/message_header_extraction_test.py ..... [ 98%]
..... [100%]

----- coverage: platform darwin, python 3.6.9-final-0 -----
Name                                                                    Stmts  Miss  Cover   Missing
-----
src/email_stream_processor/helpers/anonymization/text_anonymizer.py      40      0  100%
src/email_stream_processor/helpers/config/get_config.py                  21      0  100%
src/email_stream_processor/helpers/globals/directories.py                22      0  100%
src/email_stream_processor/helpers/globals/regex.py                      5      0  100%
src/email_stream_processor/helpers/input/input eml.py                   29      3   90%    25-27
src/email_stream_processor/helpers/input/input parquet.py                13      0  100%
src/email_stream_processor/helpers/output/output eml.py                  12      0  100%
src/email_stream_processor/helpers/output/output parquet.py              12      0  100%
src/email_stream_processor/helpers/validation/address_validation.py       12      0  100%
src/email_stream_processor/helpers/validation/text_validation.py          12      0  100%
src/email_stream_processor/parsing/message_body_extraction.py            46      0  100%
src/email_stream_processor/parsing/message_contents_extraction.py         61      8   87%    175-184
src/email_stream_processor/parsing/message_header_extraction.py           79      0  100%
-----
TOTAL                                                                    364     11   97%

===== 613 passed in 13.53s =====

```

Figure 11: Test results for event consumer

### 7.1.3 Machine Learning Jobs

While there are no traditional tests for the machine learning, static analysis and strict typing were used instead. By using these tools, the specific return types of functions could be verified, while adding type hints to help code understanding. In some ways this removed the need for bootstrap unit tests, as the execution of the jobs became much more legible and easier to follow, preventing the typical errors which unit jobs often pick up on. However, this of course does not have the added benefits that unit tests bring when refactoring code.

## 7.2 Continuous Integration

Continuous integration (CI) is a development process in which new code is commit often and tested automatically through the use of a testing pipeline. For this project, Travis CI (Travis CI GMBH, 2012) was used, which offers such a pipeline for free, configurable using a simple YAML file named “.travis.yml”. Using this pipeline, not only was the running of tests automated, but also quality tools such as Mypy (Python Foundation, 2014), to verify static typing, Pylint (Python Code Quality Authority , 2007), to adhere to a specific coding style, and Pydocstyle (Keleshev, 2014), to ensure each function was adequately commented. Taking this one step further, a pre-commit git hook framework was used to ensure code that all committed code adhered to these standards. By using these tools, the code became standardized across each of the submodules used. This CI pipeline was also used to compile the Docker files used to containerize the applications and push them to a registry to be extracted and used on deployment.

```
os: linux
dist: bionic

language: python
python:
  - 3.6

stages:
  - lint
  - test

jobs:
  include:
    - stage: lint
      install:
        - pip install pre-commit
        - pre-commit install-hooks
      script:
        - make lint






    - stage: test
      install:
        - sudo apt-get -y install libsnappy-dev

        - pip install poetry
        - poetry config virtualenvs.create false
        - poetry install
        - poetry run download_spacy_model
      script:
        - make test
```

Figure 12: Example Travis CI configuration file



## 8 System Evaluation and Experimental Results

<input type="checkbox"/> Name ^	Location	Cluster size	Total cores	Total memory
<input type="checkbox"/>  api-cluster	us-east1-b	1	1 vCPU	3.75 GB
<input type="checkbox"/>  crawler-generator-cluster	us-east1-b	1	1 vCPU	3.75 GB
<input type="checkbox"/>  kafka-cluster	us-east1-b	1	2 vCPUs	7.50 GB
<input type="checkbox"/>  streaming-cluster	us-east1-b	2	4 vCPUs	26.00 GB
<input type="checkbox"/>  tensorflow-cluster	us-east1-b	1	1 vCPU	3.75 GB

*Figure 13: Complete stack deployment machines*

### 8.1 Email Crawler

The email crawler bot performed as expected. The caveats mentioned previously did impact the overall usefulness of this crawler, however. Although the potential for the crawler was to gather 40 thousand emails per day, after processing through the pipeline, the number of useable emails deemed ready for training was only around 20 thousand. The solution to this issue would simply be to add another data source beyond this single crawler. Alternative mailing archive sites could have been used, but there could be a potential for duplicate emails being processed.

### 8.2 Kafka Storage

The Kafka stack also performed quite well. If anything, the initial outlined scaling capabilities was never required. Each of the Kafka replicas had a 100GB hard drive for storage, and the total size of emails within the entire stack never grew beyond 30GB. However, the replicas were useful to allow quicker extraction of data.

In terms of longevity of this stack, the only issue which was apparent was the cost of this solution. This is somewhat related to the above issue; in that the scale of data simply required a single replica.

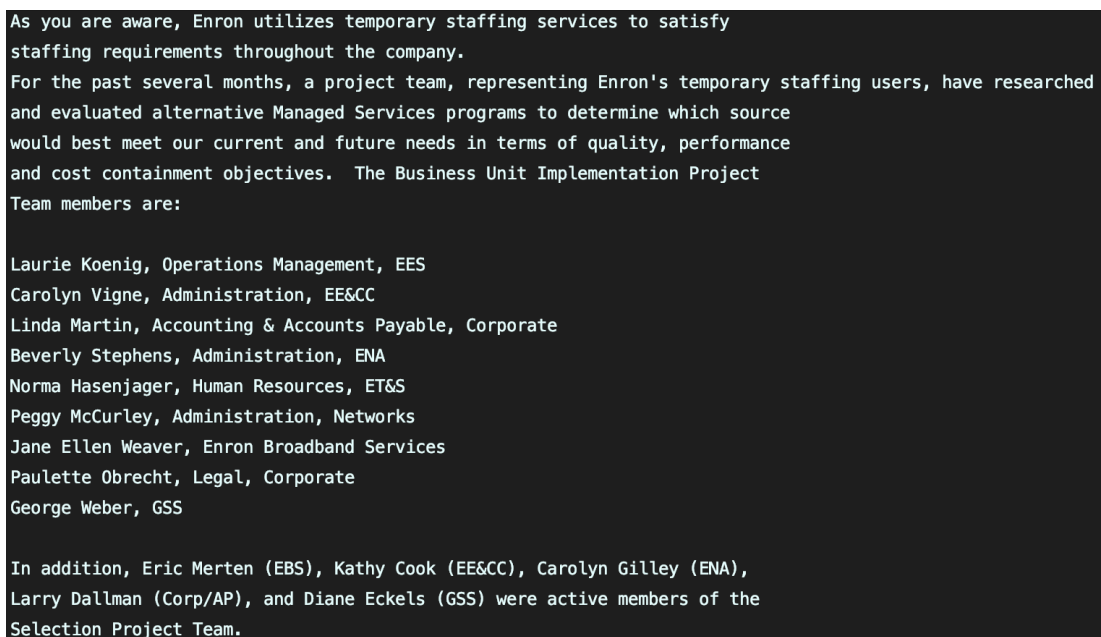
### 8.3 Spark Streaming and Batch Processing

In terms of usability and ease of setup, the Spark cluster was very simple to work with. There were some slight issues with tweaking the partition scheme of the data, as over-partitioning resulted in poor performance. Once these kinks were sorted however, the pipeline worked great. The ease of scalability came in quite useful, and through the usage of dynamic Spark executor counts, the number of nodes successfully scaled from a minimum of 1 node to a maximum of 8. This was verified through the usage of the email generator to produce an increase in events to be processed.

One issue that became apparent was the strictness of the email processing. This system immediately dropped any emails with a single defect. In reality, plenty of emails contain at least one kind of defect. On reflection, it would be smarter to ignore defects which do not pertain to the body or subject, as that is all that matters to the subsequent machine learning jobs.

### 8.4 Email Anonymization

In terms of meeting the expectations set by the requirements, this was met quite well with this solution. However, the issue that arose here is that the initial requirement and concept envisioned with processing real world data fell through, resulting in this tool being unnecessary and ultimately unused.



As you are aware, Enron utilizes temporary staffing services to satisfy staffing requirements throughout the company.

For the past several months, a project team, representing Enron's temporary staffing users, have researched and evaluated alternative Managed Services programs to determine which source would best meet our current and future needs in terms of quality, performance and cost containment objectives. The Business Unit Implementation Project Team members are:

Laurie Koenig, Operations Management, EES  
Carolyn Vigne, Administration, EE&CC  
Linda Martin, Accounting & Accounts Payable, Corporate  
Beverly Stephens, Administration, ENA  
Norma Hasenjager, Human Resources, ET&S  
Peggy McCurley, Administration, Networks  
Jane Ellen Weaver, Enron Broadband Services  
Paulette Obrecht, Legal, Corporate  
George Weber, GSS

In addition, Eric Merten (EBS), Kathy Cook (EE&CC), Carolyn Gilley (ENA), Larry Dallman (Corp/AP), and Diane Eckels (GSS) were active members of the Selection Project Team.

*Figure 14: Raw email before anonymization*

```
As you are aware, Baneddler utilizes temporary staffing services to satisfy
staffing requirements throughout the company.
For 9th January, a project team, representing Paral's temporary staffing users, have researched
and evaluated alternative Solester programs to determine which source
would best meet our current and future needs in terms of quality, performance
and cost containment objectives. The Business Unit Implementation Project
Team members are:

Mary D. Hegland, Ecoice, Genine
Nathan E. Jackson, Administration, EE&CC
Eric P. Robinson, Sitol Payable, Corporate
Brenda S. Lane, Administration, ENA
Charles R. Soto, Cabank, Branate
Gladys J. Garcia, Administration, Interetech
Tracy J. Raymo, Acuice
Lynn M. Chavez, Legal, Corporate
Felicia J. Young, Storerh

In addition, Robert J. Foreman (Viets), Michael R. Ramey (EE&CC), Ronald E. Harness (ENA),
Charollette J. Ferro (Boent), and Louise J. Villarreal (Poen) were active members of
Ganiup.
```

*Figure 15: Processed email after anonymization*

From the above, we can see the anonymization performs quite well, and for the purposes of hiding personal information, it is quite sophisticated. Issues arise with repeated company and person names not being strictly followed. Instead, each usage of a name is replaced with a new generated alternatively. If there was a requirement to continue developing this tool, this issue could be solved quite easily.

## 8.5 Topic Modelling

The topic modelling pipeline was completed quite early. As such, it received quite a long, stable training environment to continue to improve. There was issues around tweaking the learning rate, batch size and steps per training to best match the daily rate of emails, but this was resolved by experimenting with the parameters. One issue that arose as part of this experimentation is that the values are hard coded inside the Docker image. To ease the changing of parameters, these values should be either replaced with a CLI option or by using environment variables, both which can easily be changed at the K8s deployment stage. In terms of the number of topics to be discovered, 50 was settled on. This seems to have been a solid choice, but not much time was given to experimenting with other values.

As for the serving capabilities of this model, it performed perfectly. However, the performance of the API was quite slow, and could likely be improved by bundling the trained model within the Docker image, as opposed to downloading the model eternally, which results in a cold start.

```
{
  "topic": "meeting",
  "top_words": [
    "meeting",
    "go",
    "schedule",
    "look",
    "date",
    "say",
    "week",
    "time",
    "give",
    "work"
  ]
}
```

*Figure 16: Example topic modelling API response*

## 8.6 Summarization

This model was impacted quite a bit by time constraints. Due to breaking changes late into this projects schedule, the entire model needed to be trained over a short period of time. Thanks to the nature of the pipeline, it was easy to train this model over the already pre-processed data. However, there was no time to tweak any of the parameters to any real extent, beyond epochs and batch sizes.

The data trained was not optimal either. If given a second opportunity, there should be an introduction of an additional layer of filtering of emails before being processed. Working on the assumption that emails at the top of mail chain have greater relevancy in their subjects, all emails including inline messages or subject tags would be removed.

The API for this model was again great, and much more performant than the topic modelling API. This is in part due to the more lightweight model that is exported, but once again it would be benefited by bundling the model within the API docker image.

```
{
  "prediction": "meeting expert problem",
  "original": "Wednesday Meeting"
}
```

*Figure 17: Example summarization API response*

## 8.7 Model Predictions

Here are a few examples showcasing the performance of the models, using data from the Enron dataset. It is worth mentioning that these models have never encountered this data previously. The emails will be truncated to showcase the core components used when predicting.

```
Message-ID: <16020634.1075855758140.JavaMail.evans@thyme>
Date: Wed, 18 Apr 2001 02:21:00 -0700 (PDT)
From: robert.benson@enron.com
To: laura.pena@enron.com
Subject: Re: 2001 SHELL HOUSTON OPEN

Rob Benson, East power trading, Director

I would like 4 tickets for the Houston open,
If available please sent to 31st floor, location 3117d.

Thank you very much.
```

*Figure 18: Example Email #1*

**Predicted Topic:** ‘thanks’

**Predicted Summary:** ‘tickets buy’

Here we can see the predicted topic is essentially useless. This is likely due to insufficient pre-processing, as words such as ‘thanks’ and other greetings should have been removed alongside stop-words. As for the summary, there is somewhat of a relevancy to the body of the mail, but it does not include any relevant finer details.

```
Message-ID: <3777500.1075840542633.JavaMail.evans@thyme>
Date: Tue, 14 Aug 2001 07:07:00 -0700 (PDT)
From: kevin.cline@enron.com
To: doug.gilbert-smith@enron.com, seung-taek.oh@enron.com
Subject: ERCOT Zonal Load Forecasts

Starting tomorrow (8/15),
I will start publishing a load forecast for the North, West, and South zones in ERCOT.
The zonal forecasts will be run at the same time that the total ERCOT load forecast is being run.
I want to communicate how the zonal forecasts are being generated and the limitations of the zonal forecasts.

I will continue to publish a separate total ERCOT load forecast, the model of which is estimated from
1/1/1999 through 8/13/2001 (always containing the most current load data available).
The total ERCOT load forecast will differ from the sum of the zonal load forecasts due to the different
estimation periods and the use of separate models.

A comparison between the ERCOT total load forecast and the sum of the zonal load forecasts shows that they are
reasonably close.
You are welcome to do your own comparisons.

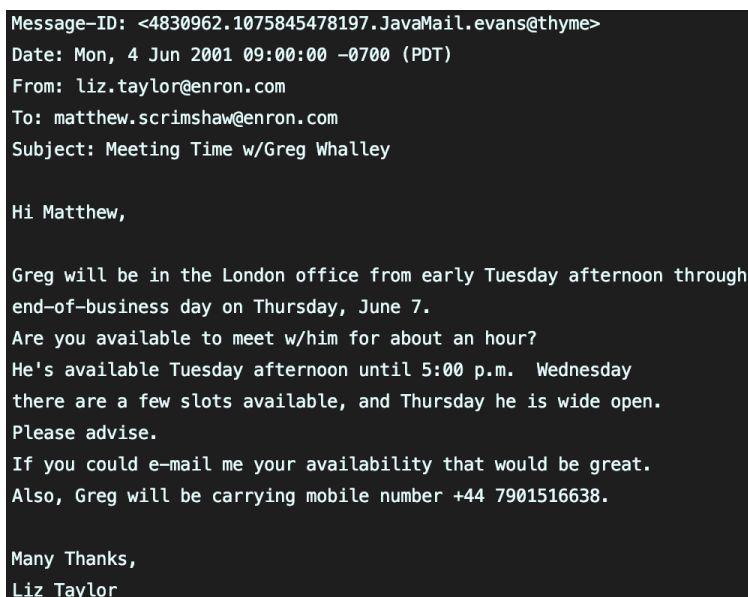
Let me know if you have any questions or comments.
```

*Figure 19: Example Email #2*

**Predicted Topic:** ‘report’

**Predicted Summary:** ‘north show end’

The predicted topic here is rather fitting, as the mail discusses forecasts and publishing, which fits well with this topic. The summary however seems rather jumbled up, and includes no reference to reports or forecasts, as would be expected

The image is a screenshot of an email message. It has a dark background with white text. The header information is at the top, followed by the body of the email. The email is from Liz Taylor to Matthew Scrimshaw, discussing a meeting with Greg Whalley.

Message-ID: <4830962.1075845478197.JavaMail.evans@thyme>  
Date: Mon, 4 Jun 2001 09:00:00 -0700 (PDT)  
From: liz.taylor@enron.com  
To: matthew.scrimshaw@enron.com  
Subject: Meeting Time w/Greg Whalley

Hi Matthew,

Greg will be in the London office from early Tuesday afternoon through end-of-business day on Thursday, June 7.

Are you available to meet w/him for about an hour?

He's available Tuesday afternoon until 5:00 p.m. Wednesday there are a few slots available, and Thursday he is wide open.

Please advise.

If you could e-mail me your availability that would be great.

Also, Greg will be carrying mobile number +44 7901516638.

Many Thanks,  
Liz Taylor

*Figure 20: Example Email #3*

**Predicted Topic:** ‘meeting’

**Predicted Summary:** ‘meeting afternoon’

Here we see the models performing at their best, with clear and mostly agreeable predictions for both the topic and summary. Realistically, this is an easy topic to detect, given its frequent occurrence in the emails the model was trained on. As for the summary, it is somewhat underwhelming to have missed the important details that said meeting is not on the current day, but other than this it performed somewhat well.

## 8.8 Deployment

Overall, the deployment tools used proved to be the correct choice for tackling the problems at hand. The sophistication of the automated deployment proved to be pivotal in terms of development speed, and also costs associated with the project. If a more traditional approach was taken, consisting of manually provisioning hardware, much more time would be wasted, and if instances were left running for longer than being worked on, the costs would be much greater also.

Keeping in the realm of cost analysis, the pipeline proved to be quite efficient in this sphere. For example, the cost of the GPUs used for training purposes range in the 7 to 8 thousand dollars range. Yet training on them only cost 50 cents per hour. This is thanks to utilizing pre-emptible resources, essentially telling Google that the resources that are requested are not going to exist for longer than 24 hours. This incurs a saving of over 50% but increases the complexity of node lifecycles.

The reliability of this pipeline however could improve in the future. The above use of pre-emptible resources can sometimes lead to issues where the requested hardware is not available, which would result in a job or service stalling and potentially failing outright. While this is always a potential problem with Cloud computing, it is especially common with pre-emptible (or ‘spot’) instances.

## 9 Closing Thoughts

While I am overall pleased with the state of the pipeline, I am underwhelmed with the trained models. As the only real deliverables of the project, they should serve as proof of the pipeline’s robustness, but as they are not up to the standard I would like, they do not do justice to the complexities of the pipeline. I feel a more experienced machine learning developer would be able to utilize this pipeline much more effectively than myself.

## 10 References

Mullan, C., n.d. *Distributed Email Pipeline*. [Online]

Available at: [https://gitlab.eecs.qub.ac.uk/40180175/distributed\\_email\\_pipeline](https://gitlab.eecs.qub.ac.uk/40180175/distributed_email_pipeline)

Alcaras, G., 2018. *Mailing List Scraper*. [Online]

Available at: <https://github.com/gaalcaras/maillingListScraper>

Alexander Sergeev, M. D. B., 2018. *Horovod: fast and easy distributed deep learning in TensorFlow*. [Online]

Available at: <https://arxiv.org/pdf/1802.05799.pdf>

Apache Software Foundation, 2008. *Apache ZooKeeper is an effort to develop and maintain an open-source server which enables highly reliable distributed coordination..* [Online]

Available at: <https://zookeeper.apache.org/>

Apache Software Foundation, 2011. *Apache Flink, Stateful Computations over Data Streams*. [Online]

Available at: <https://flink.apache.org/>

Apache Software Foundation, 2011. *Kafka, a distirbuted streaming platform..* [Online]

Available at: <https://kafka.apache.org/>

Baidu, 2017. *Baidu All Reduce*. [Online]

Available at: <https://github.com/baidu-research/baidu-allreduce>

Chen, X., 2018. *Keras Text Summarization*. [Online]

Available at: <https://github.com/chen0040/keras-text-summarization>

Cohen, W. W., 2015. *Enron Email Dataset*. [Online]

Available at: <https://www.cs.cmu.edu/~enron/>

David M. Blei, A. Y. N. M. I. J., 2003. *Latent Dirichlet Allocation*. [Online]

Available at: <http://www.jmlr.org/papers/volume3/blei03a/blei03a.pdf>



Explosion AI, 2016. *Industrial-Strength Natural Language Processing*. [Online]  
Available at: <https://spacy.io/>

Facebook, 2015. *Zstandard*. [Online]  
Available at: <https://facebook.github.io/zstd/>

Google, 2011. *Snappy, a fast compressor/decompressor..* [Online]  
Available at: <https://github.com/google/snappy>

Google, 2014. *Production-Grade Container Orchestration*. [Online]  
Available at: <https://kubernetes.io/>

Google, 20220. *Get started with Gmail: Organize your inbox*. [Online]  
Available at: <https://support.google.com/a/users/answer/9260550#2.1>

Ilya Sutskever, O. V. Q. V. L., 2014. *Sequence to Sequence Learning with Neural Networks*. [Online]  
Available at: <https://arxiv.org/pdf/1409.3215.pdf>

Keleshev, V., 2014. *pydocstyle is a static analysis tool for checking compliance with Python docstring conventions..* [Online]  
Available at: <http://www.pydocstyle.org/en/5.0.1/>

Leininger, H., 2014. *About MARC*. [Online]  
Available at: <https://marc.info/?q=about>

MailGun, 2019. *Mailgun library to extract message quotations and signatures..* [Online]  
Available at: <https://github.com/mailgun/talon>

Marmiesse, G., 2019. *Sequence to sequence example in Keras (character-level)*. [Online]  
Available at: [https://keras.io/examples/lstm\\_seq2seq/](https://keras.io/examples/lstm_seq2seq/)

Michael Chui, J. M. J. B. R. D. C. R. H. S. G. S. a. M. W., 2012. *The social economy: Unlocking value and productivity through social technologies*. [Online]  
Available at: <https://www.mckinsey.com/industries/technology-media-and-telecommunications/our-insights/the-social-economy>

Python Code Quality Authority , 2007. *Pylint, Star your Python code..* [Online]  
Available at: <https://www.pylint.org/>

Python Foundation, 2014. *Mypy is an optional static type checker for Python*. [Online]  
Available at: <http://mypy-lang.org/>

Scrapy developers, 2020. *Scrapy 2.1 documentation*. [Online]  
Available at: <https://docs.scrapy.org/en/latest/>

The Radicati Group, I., 2015. *Email Statistics Report, 2015-2019*. [Online]  
Available at: <https://www.radicati.com/wp/wp-content/uploads/2015/02/Email-Statistics-Report-2015-2019-Executive-Summary.pdf>

The TensorFlow Probability Authors, 2018. *Trains a Latent Dirichlet Allocation (LDA) model on 20 Newsgroups*. [Online]  
Available at: [https://github.com/tensorflow/probability/blob/master/tensorflow\\_probability/examples/latent\\_dirichlet\\_allocation\\_distributions.py](https://github.com/tensorflow/probability/blob/master/tensorflow_probability/examples/latent_dirichlet_allocation_distributions.py)

Travis CI GMBH, 2012. *Test and Deploy with Confidence*. [Online]  
Available at: <https://travis-ci.org/>

Twitter, Cloudera, 2013. *Apache Parquet is a columnar storage format*. [Online]  
Available at: <https://parquet.apache.org/>

Zaharia, M., 12. *Spark, lightning-fast unified analytics engine*. [Online]  
Available at: <https://spark.apache.org/>