Ke-Ming Chong

Cybercrime Chat Thread Disentanglement

Computer Science Tripos – Part II
Hughes Hall

May 10, 2024

Declaration

I, Ke-Ming Chong of Hughes Hall, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose. In preparation of this dissertation I did not use text from AI-assisted platforms generating natural language answers to user queries, including but not limited to ChatGPT. I am content for my dissertation to be made available to the students and staff of the University.

Signed Ke-Ming Chong

Date May 10, 2024

Proforma

Candidate number:	2332B
Project Title:	**Cybercrime Chat Thread Disentanglement**
Examination:	**Computer Science Tripos – Part II, 2024**
Word Count:	11977[1]
Code Line Count:	3921[2]
Project Originator:	Dr. Jack Hughes
Project Supervisor:	Prof. Alice Hutchings and Dr. Jack Hughes

Original Aims of the Project

The original aim of this project was to implement and compare different models for classifying messages from cybercrime chats into their respective threads. The models intended to be created were: Support Vector Machines (SVM) and Bidirectional Encoder Representations from Transformers (BERT), with Gradient Boosting (GradBoost) as an extension. This also included data exploration and pre-processing, as well as feature engineering and implementing different representations for features.

Work Completed

All of the core and extension success criteria were met. Extensions included exploring Gradient Boosting as a viable model, as well as feature engineering. As models from the original project did not produce satisfactory results, the project was further extended to deal with new threads and unseen creators. All models were implemented, evaluated and compared.

Special Difficulties

None.

[1]Computed using TeXcount web service, including tables, captions, headers. Source: https://app.uio.no/ifi/texcount/online.php

[2]Computed using cloc. Source: https://github.com/AlDanial/cloc

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Cybercrime encompasses a wide range of illicit activities including hacking, phishing and distributed denial of service attacks [1]. The surge in cybercrime incidents globally [2,3] highlights the need to understand and address this threat [1]. In recent years, the cybercrime landscape has witnessed a significant shift away from traditional underground forums towards messaging platforms such as Telegram and Discord [4, 5]. This is unsurprising given Telegram's commitment to user safety through end-to-end encryption in its secret chat option [6], allowing criminals to communicate without administrator oversight [7]. Moreover, Discord [8, 9] is steadily emerging as the preferred platform for the growing younger demographic within the cybercrime community [10]. The adoption of messaging platforms allows illicit discussions to occur not only on the Deep Web and Darknet [11] but also on the Surface Web [12], broadening criminals' reach to a wider audience [13, 14]. Criminals may create groups and servers to share various illicit material, from leaked passwords [15] to hacking tutorials [8], or even use the platform as a command-and-control channel for malware [16]. Unlike in forums, criminals could use bots [17] and different usernames across different chats and servers, further complicating detection.

Forums organise discussions into threads, facilitating research as analysis can be done at the thread level [18, 19]. In contrast, messaging platforms lack threaded conversations, making it harder for researchers to discern when a discussion starts or stops, especially when multiple discussions overlap [20]. Within a single group chat, many simultaneous and distinct conversations may occur concurrently. Elsner and Charniak's [21] research on disentangling Internet Relay Chat dialogue revealed that an average of 2.75 simultaneous threads are active at a time. This can result a conversation that does not form a "contiguous segment of the chatroom transcript, but is frequently broken up by interposed utterances from other conversations". Table 1.1 shows an excerpt from one of the boards where multiple threads are interleaved within the same conversation. Disentangling interwoven threads is a non-trivial and tedious task, especially if done manually [21, 22]. Previous efforts to address this issue have employed deep learning methods but only using data from messaging platforms for training [23, 24], rather than forums. Existing research on cybercrime chats exists for in-depth analysis through Wordnet [25] and text mining [26] techniques, but not with the models used in this project.

1.2 Project aim

This project aims to evaluate and compare various machine learning models and their abilities to accurately identify and disentangle different 'threads' within cybercrime conversations, mirroring the structure found in existing forums. Leveraging on forum data where threads are

Creator	Message	Thread	Date
Creator A	If you would do i right it would end up with 0	Thread 1	7/10/19 10:59
Creator B	Pretty cool like technique.	Thread 2	7/10/19 12:52
Creator C	Have you seen video idiot ? Awesome share again	Thread 2	7/10/19 13:24
Creator D	I tried this with Windows defender, Nod32, Avira AND AVG, and it bypassed all of them....	Thread 2	7/10/19 13:34
Creator E	V3 is in the works lol. whoz excited :3	Thread 1	7/10/19 19:09
Creator F	Maybe, are you ditching JQuery ?	Thread 1	7/10/19 19:11
Creator E	to what alternative?	Thread 1	7/10/19 19:13
Creator D	No problem, happy to help	Thread 2	7/10/19 19:40

Table 1.1: Example message board conversation

separated, this project seeks to help cybercrime researchers better understand and measure topics of conversations discussed on messaging platforms more efficiently.

1.3 Project summary

In this dissertation, I present:

- Implementations and evaluations of SVM, BERT and gradient boosting models for classifying messages into threads. (Phase 1)
- A new implementation approach to the task, offering a fresh perspective. (Phase 2)
- Comparisons between the aforementioned models.

This project comprised two distinct phases. Phase 1, the initial version, investigated whether a complex neural network could outperform a simpler machine learning model in classifying messages into threads. Phase 2, a subsequent iteration, explored the inclusion of new threads and unseen creators in the classification process.

1.4 Ethics

Unavoidably, working with cybercrime-related data may raise some ethical concerns [27]. Before gaining access to the CrimeBB dataset, an ethics review was conducted by the department, ensuring that crime against vulnerable individuals was reported and data was anonymised and stored securely. All outputs were only printed on local machines and no identifiable information was committed to GitLab by using `gitignore`. Obtaining informed consent for all members of the forums, though important, was not achieved as contacting all users would be considered spamming [28]. Publicly accessible messages were analysed as a whole rather than identifying individual users, which exempted the need for informed consent according to the British Society of Criminology [29].

Chapter 2

Preparation

This chapter includes the project starting point (Section 2.1), introduces the dataset (Section 2.2), background theory required for the SVM, BERT and gradient boosting models (Section 2.3), followed by the languages and systems used (Section 2.4) and the requirements analysis (Section 2.5).

2.1 Starting point

This project used Python and its libraries for all models and evaluations. I have experience with Python from the Scientific Computing and Machine Learning and Real-world Data courses in IA, Data Science course and group project in IB, and personal projects. This year, I took the Natural Language Processing and Deep Neural Networks modules.

2.2 CrimeBB

The data utilised for this project was retrieved from the Cambridge Cybercrime Centre's CrimeBB dataset, which was created by web crawling underground forums and messaging platforms [30]. These forums are online communities that focus on illicit activities [31]. A significant portion of the data comes from Hack Forums [32] which is accessible via the Surface Web. Hack Forums gained notoriety in 2016 when the Mirai botnet [33] was leaked on the platform, spawning many copycat distributed denial-of-service attacks [34, 35]. Additionally, Hack Forums also advertised botnets and tools for stealing data [36].

2.2.1 Dataset

The database consists of 4,321 boards across 38 sites. This project used the **Hacking Tutorials** (797 messages, 17 threads) and **Cryptography** (3,416 messages, 63 threads) boards from Hack Forums as the main datasets, seen in Tables 2.2 and 2.3. **Hacking Combined** was a synthesised dataset combining various hacking-related boards and was used for BERT pre-training, seen in Section 3.6.3. Using the Hacking Tutorials board initially was advantageous due to its long messages and relatively small dataset size, facilitating model development and testing. I chose the Cryptography board for evaluation due to its technical language and moderate amount of numerical data. Dataset formats are visualised in Table 2.1a.

Data from Hack Forums was specifically selected due to its extensive collection of English-language boards. To extract relevant datasets, SQL queries were crafted to retrieve messages from these boards. Only messages posted after 2018 were selected to ensure relevance, aligning with best practices to avoid using outdated information [37]. The longer messages in chosen datasets contributed to a richer corpus for the models to learn from [38].

Data from cybercrime-related Discord servers were also obtained from the Cambridge Cybercrime Centre, with many of these servers having short conversations. This database contains 430 unique servers with topics ranging from politics to hacking, their format shown in Table 2.1b. A server for hacking support was prioritised as the conversation lengths were between 500 to 2,000 words, and the topic was suitable and similar to Hacking Tutorials. This project used this chat data for evaluation of the models as an extension in Section 4.7.

Column	Description
id	ID of post
board_id	ID of the forum board
thread_id	ID of thread within board
creator_id	ID of post creator
content	Textual content of post
created_on	Post creation date

(a) Forum dataset format

Column	Description
id	ID of message
user_id	ID of message creator
channel_id	ID of channel within discord server
content	Textual content of message
timestamp	Message sent date

(b) Discord dataset format

Table 2.1: Dataset formats

Boards	Dataset Name	Number of Messages	Number of Threads	Number of Creators
Hacking Tutorials	Hacking Tutorials	797	17	485
Cryptography	Cryptography	3416	63	1369
Beginner Hacking E Book Hacking Hacking Requests Website Hacking Wi-Fi Hacking	Hacking Combined	3123	69	1574

Table 2.2: Boards used from CrimeBB database

Dataset	Hacking Tutorials	Cryptography
Number of Messages	797	3416
Number of Threads	17	63
Mean Message Length	38 Words	25 Words
Median Message Length	13 Words	13 Words
Longest Message	3143 Words	1680 Words
Shortest Message	1 Word	1 Word
Longest Thread	113 Messages	271 Messages
Shortest Thread	26 Messages	26 Messages

Table 2.3: Dataset summary

2.3 Background and theory

As much of the project was implemented using Python libraries, background information is provided for context.

2.3.1 Support vector machines

This project used SVMs as one of the main models. It is a machine learning algorithm widely used for classification tasks, constructing a hyper-plane in a high-dimensional space which optimises the margin between support vectors or classes [39]. Support vectors refer to the datapoints lying closest to the boundary between two classes. After all datapoints have been plotted in n dimensions, the SVM finds the $n-1$ dimensional hyperplane such that its distance to the support vectors in each class is the largest. This project used soft-margin classification [40], handled by regularisation parameter C which is a hyperparameter optimised in Section 4.5.1.

2.3.2 Neural networks and deep learning

Neural networks are a type of machine learning process, known as deep learning, that uses interconnected layers of artificial neurons to make predictions or decisions [41]. A simple representation is as follows:

$$f(x) = W_n\phi...\phi(W_2\phi(W_1x + b_1) + b_2) + ... + b_n \tag{2.1}$$

where W_i and b_i represent weight and bias matrices respectively, ϕ the activation function and x the input vector.

Activation functions

Activation functions in neural networks introduce non-linearity, allowing complex patterns to be learnt. They calculate the output of a node based on inputs to the function and weights of the node, seen in Equation 2.1. For input vector x, some common activation functions are as follows:

The **ReLU** activation function converts all negative values to 0:

$$\phi(x) = max(0, x) \tag{2.2}$$

The **Tanh** activation function is defined as:

$$\phi(x) = tanh(x) \tag{2.3}$$

These were used for custom classification heads for BERT, explained in Section 3.6.5. In a multilayer perceptron (MLP), the activation function is typically inserted between two linear layers, enabling the network to learn complex mappings from input to output. This project used MLP to combine textual, categorical and numerical data, seen in Section 3.6.4.

Loss functions

A loss function, often **cross entropy loss** for classification, is used to indicate how far the model's predicted outputs are from the actual labels. When training, the aim is to minimise this value of loss. Cross entropy measures the difference between two probability distributions: predicted probabilities and true distribution of labels.

$$L_{CE} = \sum_{i=1}^{n} y_i log(p_i) \tag{2.4}$$

where n is the number of classes, y_i the actual label of class i and p_i the probability of predicting class i. This project used cross entropy loss for all BERT models.

Optimisation

Lastly, at every step, the model's parameters are optimised with respect to loss. Adaptive Moment Estimation (**Adam**) [42] is widely used for BERT and other large-scale models due to its ability to adapt learning rates. It combines two extensions of stochastic gradient descent: Adaptive Gradient Algorithm and Root Mean Square Propagation [42]. Adam makes use of momentum, which adds a fraction of the previously calculated gradient to the current one, speeding up convergence by maintaining momentum and aiding in escaping local minima.

$$m_{t+1} = \beta_1 m_t + (1 - \beta_1) g_t \tag{2.5}$$

$$v_{t+1} = \beta_2 v_t + (1 - \beta_2) g_t^2 \tag{2.6}$$

$$\theta_{t+1} = \theta_t - \eta \frac{m_t}{\sqrt{v_t + \epsilon}} \tag{2.7}$$

where m and v represent the exponentially decaying average of the gradients and squared gradients respectively, β the exponential decay rate, g the gradient, η the learning rate and θ the parameter being updated. This project used Adam to optimise model parameters.

2.3.3 BERT

This project used BERT as the other main model. It is a language representation model trained on the Toronto BookCorpus (800M Words) and English Wikipedia (1500 Words) for masked language modelling and next sentence prediction [43]. An encoder-only transformer, it uses contextual embeddings generated from **self-attention** [44] that considers the entire sequence bidirectionally. In self-attention, each word is converted into a vector representation. Then, three vectors are derived: **query**, **key** and **value**, obtained through linear transformations. The attention score is calculated as the dot product of query and key vectors, determining the focus of one word on the other. BERT employs multiple sets of these transformations, known as attention heads. This allows the semantic and syntactic relationships between words within a sequence to be fully understood. Moreover, BERT can be fine-tuned on different NLP tasks, including classification [45].

The implementation used for this project, $BERT_{BASE}$, has 12 encoders with 12 bidirectional self-attention heads totalling 110 million parameters. This project briefly explored other BERT-like models such as A Lite BERT (ALBERT) [46], which reduces parameters for faster training, and Robustly Optimised BERT Pretraining Approach (roBERTa) [47], which modifies key hyperparameters in BERT and omits the next-sentence prediction pretraining objective.

2.3.4 Tokenisation

Textual data has to be tokenised before feeding them into models. Tokenisation is a preprocessing step that splits text into a sequence of tokens, simultaneously creating a vocabulary that is derived exclusively from the training set. Lemmatisation is similar, but only the root word is kept, resulting in a smaller vocabulary and a more generalised dataset. Both tokenisation and lemmatisation were explored in this project.

BERT tokeniser

BERT employs WordPiece tokenisation [48], splitting text into subwords, allowing for more fine-grained representation [43]. The BERT tokeniser breaks down a sequence into tokens, assigning each token a corresponding integer from its vocabulary. Additionally, special tokens are added: [CLS] at the sequence's beginning and [SEP] at the end, and [PAD] for padding to the maximum sequence length of 512 tokens. These tokens are crucial for delineating the sequence's boundaries, especially in tasks involving multiple sequences.

Moreover, the tokeniser provides token type IDs and an attention mask. Token type IDs indicate the position of each sequence in the input, primarily used when dealing with paired input sequences. It assigns '0's to tokens in the first sequence and '1's to those in the second. Meanwhile, the attention mask helps identify which token positions to focus on, particularly when sequences have padding to accommodate varying lengths.

2.3.5 Vectorisation

After tokenisation, the text has to be vectorised and transformed into a format compatible with machine learning models. Vectorisation converts textual tokens into numerical vectors. BERT tokeniser does this automatically, but it has to be done separately for other models. These vectors are concatenated with other categorical and numerical features to form feature vectors as input to the models. Vectorisation methods used in this project are **bag of words**, **n-grams** and **TF-IDF**. Bag of words represents text data by counting the occurrence of each word in a sequence. This can be extended to n-grams, where sequences are tokenised in groups of n instead.

Term frequency - inverse document frequency

Term frequency - inverse document frequency (TF-IDF) measures the significance of a word within a document, considering both its frequency and rarity across all documents. The TF-IDF score of a term is calculated by multiplying two distinct metrics - the term frequency and inverse document frequency. A term has high importance and TF-IDF score if it occurs often in one document, but rarely in others.

Term frequency of a word in a document is calculated by counting the raw number of occurrences of the word in the document. Normalisation may be applied by dividing by the total number of terms in the document, mitigating the influence of document length. **Inverse document frequency** is a measure of how much information a term provides, with rarer terms providing more information than common ones. It is the logarithmically scaled inverse fraction of the documents containing the term.

2.3.6 Fine-tuning

While BERT was pre-trained on a large text corpus from diverse sources, it may not be directly applicable to all downstream NLP tasks like classification. Fine-tuning allows BERT to learn task-specific patterns and nuances from the target dataset [45], leveraging on its pre-trained knowledge of language and semantics while capitalising on new contextual knowledge. Fine-tuning includes pre-training on **in-domain** and **cross-domain data**, **truncation**, and **discriminative fine-tuning**.

In-domain and cross-domain pre-training involve training the model on text from different sources, improving the model's vocabulary and semantic understanding. Truncation focuses on retaining the most relevant parts of a sequence that exceeds the maximum of 512 tokens.

Discriminative fine-tuning

Discriminative fine-tuning involves splitting the model into its constituent layers and applying different learning rates to each layer or group of layers [49]. This aims to optimise performance by allowing small adjustments to the parameters of different layers, tailoring to the target task.

2.3.7 Gradient boosting

Gradient boosting is an ensemble technique that combines predictions from multiple weak learners, each slightly better than random predictors, into a stronger learner [50]. Decision trees are often chosen as weak learners due to their versatility across datasets and classification tasks [51]. Like neural networks, gradient boosting relies on optimising a loss function. Binary cross entropy was used for this project as default from the scikit-learn library [52].

$$L_{BCE} = -\frac{1}{N}\sum_{i}^{N}[y_i ln(p_i) + (1 - y_i) ln(1 - p_i)] \tag{2.8}$$

2.4 Languages and systems

This project was implemented using **Python** [53], leveraging its extensive libraries for building, training and evaluating machine learning models, as well as for visualisation. Key libraries included **pandas** [54] for dataset handling, **NumPy** [55] for computations and **Matplotlib** [56] for result visualisations. This project employed the **scikit-learn** [52] library for SVM and gradient boosting, and **PyTorch** [57] for BERT to match the Hugging Face [58] implementation and align with the Part II Deep Neural Networks module. This project utilised the **transformers** library [59], part of the Hugging Face ecosystem, to access the pre-trained BERT model. To alter and augment data for testing model robustness, this project used the **nlpaug** library [60].

Due to the sensitive nature of the dataset, all work was conducted on a personal laptop with Visual Studio Code [61] as the integrated development environment. I used `Git` for version control, with the project backed up regularly to a private GitLab [62] repository. Strict `gitignore` rules ensured that no figures or data were pushed to GitLab.

2.5 Requirements analysis

The requirements are largely unchanged from the project proposal. However, after considering the available datasets and initial models, a new approach termed **Phase 2** was developed to deal with unseen creators and new threads, such as in unlabelled chat data. This still meets the criteria of core work mentioned in the project proposal, but I have added an extension to test these models with Discord data.

Core work

- Develop, train and evaluate SVM and BERT on the CrimeBB dataset to classify messages into threads.

Extensions

- Explored how feature engineering may aid in making better models, through features inherent in each message.
- Explored the feasibility of using gradient boosting as a model for classification.
- Further to these extensions, this project used Discord data as input to the built models for evaluation.

Chapter 3

Implementation

This chapter details the implementation of the project, explaining software engineering techniques (Section 3.1), describing the datasets (Section 3.2), and models used. Model implementations were split into two phases, with Phase 1 (Sections 3.4, 3.5, 3.6, 3.7) implementing different classifiers, and Phase 2 (Sections 3.8) implementing a new approach to the task.

3.1 Software engineering

3.1.1 Repository overview

```
src/
├── data/
│   ├── hacking_tutorials.csv
│   ├── cryptography.csv
│   ├── hacking_combined.csv
│   ├── discord_test_hacking.csv
│   └── ...
├── preparation/
│   ├── preprocess.py ........ Functions for processing datasets: 201 Lines
│   ├── synthesise.py ........ Functions for synthesising datasets: 105 Lines
│   └── interleave.py ........ Functions for interleaving threads: 80 Lines
├── models/
│   ├── svm.py ........ Functions for testing SVM features: 209 Lines
│   ├── svm_models.py ........ Model definition for SVM: 281 Lines
│   ├── bert.py ........ Functions for testing BERT features: 37 Lines
│   ├── bert_models.py ........ Model definition for BERT: 640 Lines
│   ├── bert_helpers.py ........ Helper functions for BERT: 501 Lines
│   ├── gradboost.py ........ Functions for testing gradient boosting features: 152 Lines
│   ├── gradboost_models.py ........ Model definition for gradient boosting: 226 Lines
│   ├── testing_framework.py ........ Functions for testing and evaluating models: 170 Lines
│   ├── phase2_svm.py ........ Model definition for Phase 2 SVM: 471 Lines
│   ├── phase2_bert.py ........ Model definition for Phase 2 BERT: 518 Lines
│   ├── phase2_eval.py ........ Functions for evaluating Phase 2 models: 178 Lines
│   ├── visualise.py ........ Functions to visualise results and threads: 129 Lines
│   └── ...
└── figures/
```

3.1.2 Project management and methodology

This project used the **Agile** methodology [63], which prioritises continuous improvement. Each model went through an iterative process of planning, designing, building, testing, evaluating and then improving. After each model was built, they were then compared and evaluated on the same dataset. This project was split into two phases. **Phase 1** was the original part of the project where models were built and evaluated. **Phase 2** implemented an improved algorithm due to the shortcomings of Phase 1 and dealt with unseen creators and new threads. To evaluate the models, I implemented a **testing framework** for machine learning models that involved checking the data and perturbing data to test model robustness. I employed **object-oriented programming** techniques to develop the models and incorporated **docstrings** to follow best practices by providing comprehensive comments and descriptions.

3.2 Data exploration

The first step of the project after gaining access to the dataset was to explore and identify potential features which could be used to create more efficient and effective classification models. I used the smaller Hacking Tutorials dataset for training, debugging, fine-tuning and testing different methods on the models, before then using the larger Cryptography dataset, explained in Section 2.2.1.

3.2.1 Initial insights

The datasets used are obtained from the CrimeBB database and cover significant cybercrime topics such as hacking and cryptography, as explained in Section 2.2. I first required an understanding of the distribution of messages and threads within the dataset.

Class imbalances

The datasets exhibited imbalances in the distribution of messages, where some threads had significantly more messages than others, indicated longer conversations. Most threads had few messages and creators, with no visible correlation between number of messages per thread and number of creators per thread, shown in Figures 3.1 and 3.2. Figures 3.3 and 3.4 are histograms showing the distributions of threads based on the number of messages and creators they contained, respectively. Failing to balance the data would cause the models to spend most of their time learning from the majority class and not enough from the minority class, unable to generalise well to unseen data [64].

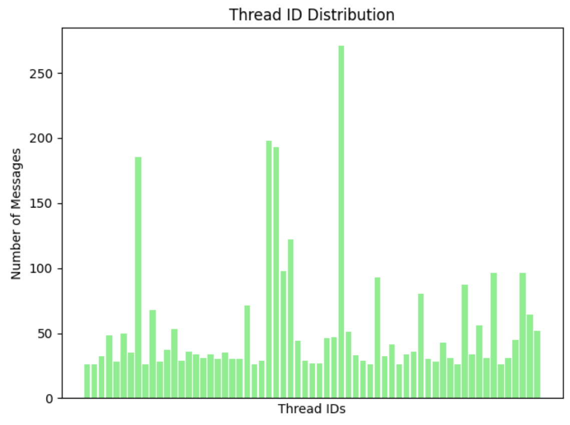


Figure 3.1: Messages per thread

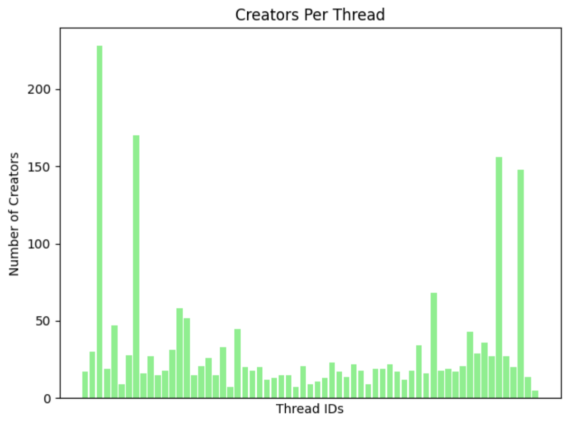


Figure 3.2: Creators per thread

Creators

Creators refer to the provenance or account that authored the message, an important feature, represented in the dataset by `creator_id`. A vast majority of creators only participated in a single thread, while a very small number participated in more, seen in Figure 3.5.

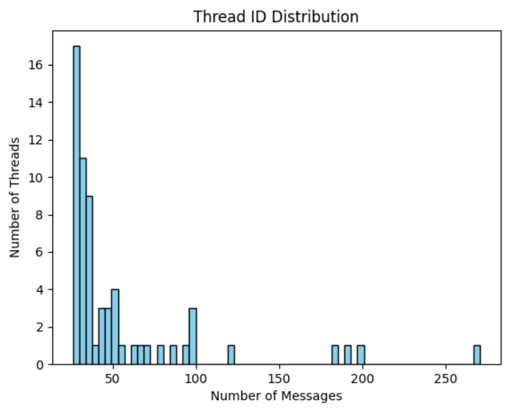


Figure 3.3: Message distribution

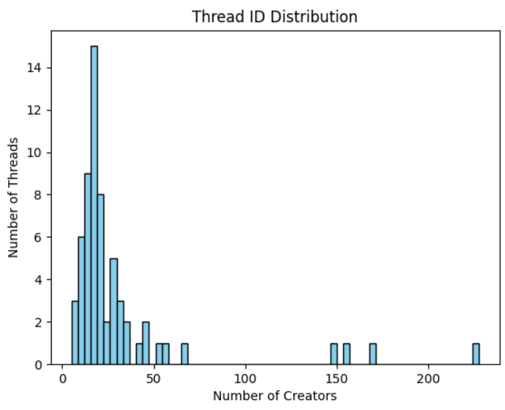


Figure 3.4: Creator distribution

Links

Some messages contained links leading to other posts within the same thread, indicating that the current post was a reply to the post being linked to. These links were useful as messaging platforms offer reply features, and leveraging these replies helped with thread classification. Post IDs were important for identifying the message being replied to, thus they were extracted before removing links from the text in the preprocessing stage. Table 3.1 shows the format of links in the dataset.

Description	Format
Replies	***CITING*** [https://(*forum site*)/showthread.php?pid=(*post id*)#pid(*post id*)] ***CITING***
Images	***IMG*** [https://(*forum site*)/images/smilies/(*image file*)] ***IMG***
External Links	***LINK*** [https://(*external site*)] ***LINK***

Table 3.1: Link formats

Emojis

Some messages contained emojis, including both graphical emojis and emoticons created using punctuation. Most creators did not use emojis or emoticons in their messages, shown in Figure 3.6. Few threads contained emojis, while emoticons were more frequently used though typically numbering less than 15 per thread, shown in Figure 3.7. This allowed emojis and emoticons to be distinctive features for certain threads, helping with classification.

[H]

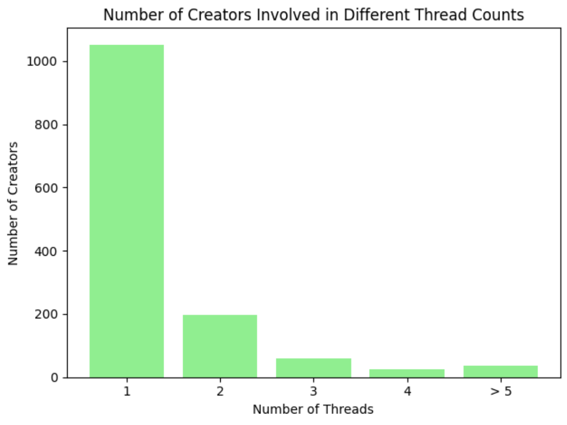


Figure 3.5: Creators involved in different thread counts

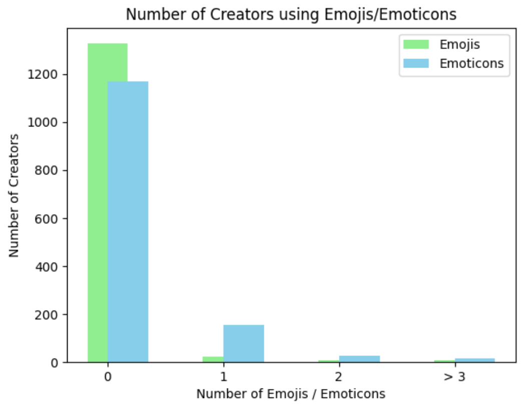
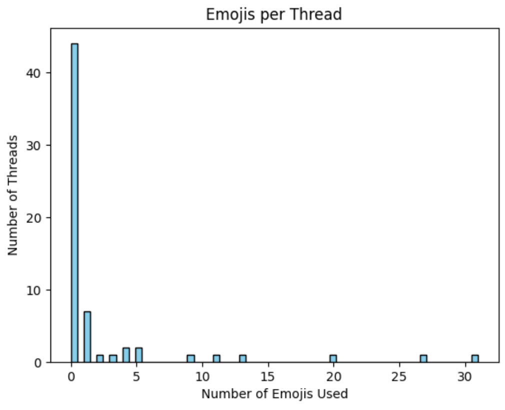
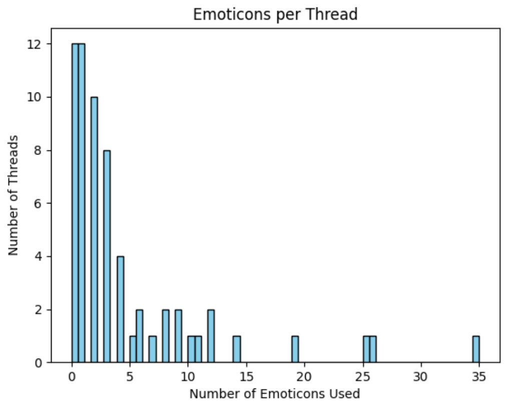


Figure 3.6: Emojis/emoticons used by creators



(a) Emojis used per thread



(b) Emoticons used per thread

Figure 3.7: Number of emojis / emoticons used per thread

3.2.2 Feature exploration

Apart from text, each message had various features that could be useful for classification.

Creator IDs

Creator ID (CID) of each message was the main categorical feature, obtained from the `creator_id` field in the dataset. As mentioned in Section 3.2.1, creators were an important feature for classification. Moreover, some messages served as replies to other messages in the same board, potentially authored by a different creator. These messages were identified through links in the text, mentioned in Section 3.2.1, and were termed 'linked messages' in this project. As each linked message may have a different creator, the CIDs of the current message and all its linked messages were used as features, seen in Equation 3.1.

$$cid_m = \left(\bigcup_{n \in replies\_to(m)} n.cid \right) \cup m.cid \tag{3.1}$$

To access the CID of linked messages, the link in the message was extracted and used to index the dataset to find the linked message. The CID was then extracted from this message. This

Col 1	Col 2	Col 3
A	C	0
E	D	0
E	0	0
B	C	D
B	E	A

(a) Padded list representation for creators A-E

A	B	C	D	E
1	0	1	0	0
0	0	0	1	1
0	0	0	0	1
0	1	1	1	0
1	1	0	1	0

(b) One-hot encoded representation for creators A-E

Table 3.2: Different representations for same CID sets

project experimented with two representations of CIDs: **padded list** and **one-hot encoding**.

Padded list included all CIDs present in a message's CID set, padded with '0's to match the largest set in the dataset. Meanwhile, the length of each one-hot encoded list was equal to the number of unique CIDs in the dataset, with each CID allocated an index within this list. Each CID in a message's CID set was encoded with '1's at their respective indices and '0's everywhere else. This way, the machine learning models did not treat the numerical values of CIDs as numerical data, but as categorical data instead. This was important as there was no quantitative relationship between two different CIDs. Table 3.2 visualises these representations.

Dates and times

As conventional messaging platforms exhibit sequential interactions where messages are composed based on preceding ones, understanding the temporal relationships and proximity between messages was beneficial for classification. Representations used were **timestamps**, **offsets** and **one-hot encoding**.

The **timestamps** utilised were synthesised after the preprocessing stage and were directly employed as features. Subtracting the timestamp of the first message in the board from the timestamp of each subsequent message gave the **offsets**. Furthermore, various temporal components such as hour, day, month, year and day of the week were extracted from timestamps and **one-hot encoded**.

3.2.3 Extension: feature engineering

Apart from data already present in the dataset, messages inherently contained other information that were used as features.

Part of speech tags

The spaCy [65] tokeniser was used to extract the part of speech (POS) tag for each token using the `pos_` attribute. The frequency of each POS tag within each message was compiled, scaled and used as features. Upon reflection, focusing on the semantic content of words instead of grammatical types was more suitable for the project's objectives. Thus, it was more appropriate to classify messages based on similarities in topics and content instead of POS tags.

Emojis

The presence of emojis and emoticons could improve classification given their scarcity in most threads, seen in Figures 3.7a and 3.7b. However, conventional vectorisers used in this project, such as `TfidfVectorizer`, do not accommodate emojis and emoticons, typically removing them and only keeping alphanumeric data. To address this, the presence of emojis and emoticons were encoded as binary features.

3.3 Data preparation

3.3.1 Data preprocessing

The primary feature utilised for classification was the textual content of messages, which required preprocessing to render it both useful and usable by machine learning models. While these steps may potentially enhance classification results, they carried the risk of inadvertently removing valuable information essential for classification. The preprocessing techniques employed in this project were: **drop NaNs**, **link removal**, **stop word removal**, **non-alphanumeric character removal**, **text filtering** and **spell checking**.

As all useful information from links had already been extracted, they, along with NaN values, no longer offered significant value for classification. The `dropna()` function from pandas [54] was used to remove NaN values as they could not be reasonably imputed or replaced for this task. Regular expression functions `sub()` and `replace()` removed links.

Stop word removal

Words that occurred frequently, but carried little to no meaning or importance, were filtered from the messages before classification. This placed greater emphasis on words that provided significant contextual information. However, removal of stop words potentially had a negative effect on the results by altering the original meaning or context of the text**??**. The NLTK library's [66] stop word corpus was used to implement stop word removal.

Non-alphanumeric character removal

Some texts contained unwanted symbols that made tokenisation and classification challenging, particularly when these symbols lacked meaningful information. The built-in `isalnum()` function was used to filter out these symbols. Though this did remove emojis and emoticons, the binary feature representation in Section 3.2.3 continued to capture their presence.

Text filtering

The aim of this process was to eliminate outliers from the dataset that could potentially affect model performance. Two approaches were considered: filtering texts based on length or frequency of words. The former was achieved by only keeping texts within the interquartile range, eliminating excessively long or short messages. The latter involved removing words that were infrequent in the corpus. However, these methods of removing outliers potentially led to information loss and bias, thus they were not used [67].

Spell checking

Text messages often contained typos and misspellings which were treated as new words with few occurrences when tokenised, leading to poor classification results. To address this, the pyspellchecker library [68], which selects the most likely corrections based on word frequency, was used. However, this library had a limited vocabulary which did not encompass proper nouns and slang words, replacing them and thus removing significant meaning and information from the messages. Thus, this method was not adopted for the final models.

Lemmatisation

Similar to tokenisation, where a text is converted into tokens, lemmatisation extracts the base forms of each word instead of merely segmenting the text. This reduced the number of tokens, consequently reducing the complexity of each feature vector and made classification faster in the process [69]. Using the spaCy library [65], the `lemma_` attribute of each token was accessed after tokenisation. While lemmatisation could help improve generalisation by simplifying the feature space, some information and context was lost [69].

3.3.2 Class imbalances

As stated in Section 3.2.1, the datasets exhibited class imbalances where some threads had significantly more messages than others. If not addressed, imbalances could result in the models exhibiting bias towards the majority class, classifying more datapoints into the majority class and generalising poorly to unseen data [64], seen in Figure 3.8. This was managed in a few methods: **stratified train-test split**, **Synthetic Minority Oversampling Technique (SMOTE)** [70] and **duplicate oversampling**.

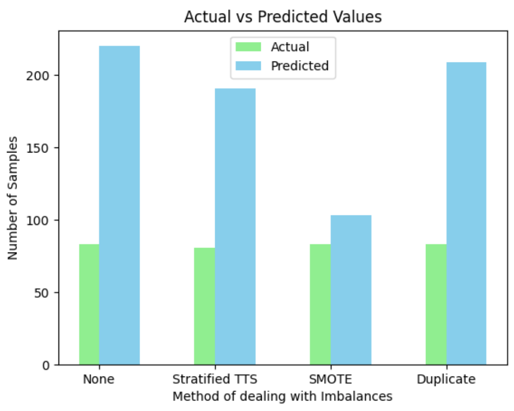


Figure 3.8: Actual vs predicted datapoints in majority class

Stratified train-test split

Before training, scikit-learn's [52] `train_test_split()` function was used to split the dataset into train, validation and test sets. By setting the `stratify` parameter in `train_test_split()` to the list of labels, data was partitioned into train and test sets while maintaining the class distribution in both sets. This prevented instances where either set had over-representations of the majority class or lacked representations of the minority classes.

Duplicate oversampling

Messages from the minority classes were duplicated until a certain ratio with respect to the majority class was reached. However, this method had potential drawbacks such as increased risk of overfitting and loss of information due to reduced diversity, affecting the model's ability to generalise. This was implemented by randomly selecting messages from the minority classes and duplicating them until there were half as many instances per class compared to the majority class

Synthetic Minority Oversampling Technique (SMOTE)

SMOTE, implemented with the imbalanced-learn library' [71] `SMOTE` class, drew lines from k samples within the feature space and generated new samples at points along those lines, synthesising new datapoints for the minority classes. The `k_neighbors` parameter determined the number of neighbours to consider when generating these synthetic samples. This project used a value of `k_neighbors` = 5.

3.3.3 Time series split

This project explored an alternative method for capturing the time series relationships between messages as the initial approach in Section 3.2.2 did not produce satisfactory results. Scikit-learn's [52] `TimeSeriesSplit` class segmented the data into n folds where the first fold was allocated for training and the second for validation. Subsequently in each iteration, both folds were allocated for training and the third for validation. This was done until the n^{th} fold was used for validation. This maintained the time series structure of the dataset. The `n_splits` parameter determined the number of folds to segment the data into.

3.4 Phase 1 overview

Phase 1 of the project involved first synthesising the preprocessed dataset to create dates, allowing the messages to seem akin to that of a conversation from messaging platforms. Subsequently, the dataset was partitioned into train and test sets before being fed into machine learning models for classification. The primary objective of this phase was to leverage on supervised learning techniques to classify messages into their respective threads based on information contained within each message. Refer to Figure 3.9 for the flow diagram.

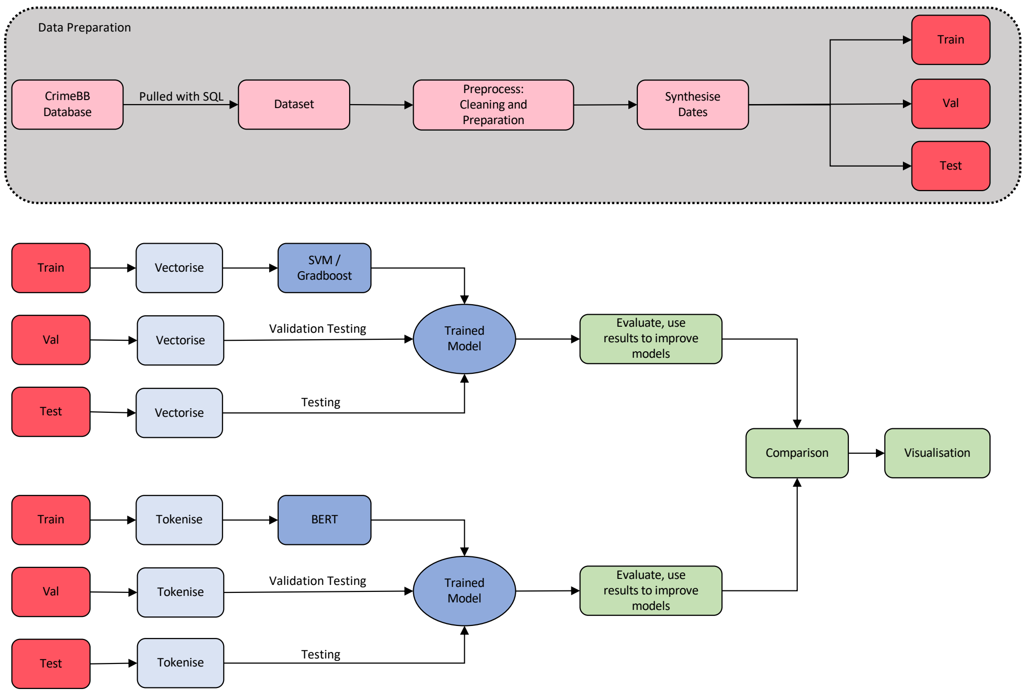


Figure 3.9: Phase 1 flow diagram

3.4.1 Synthesising

Given that data sourced from forums exhibited vast date and time differences between threads, they had to be transformed to mirror the conversational flow typical of messaging platforms. This transformation involved interleaving threads while preserving the chronological sequence of messages within each thread, followed by assigning new timestamps to each message. A time window of one month was selected, and random timestamps were generated within this window for each thread. These timestamps were sorted and assigned to each message within their respective threads. Figure 3.10 shows how threads are interleaved randomly. Phase 2 adopted a refined synthesis method to better emulate the structure of messaging platforms, seen in Section 3.8.1.

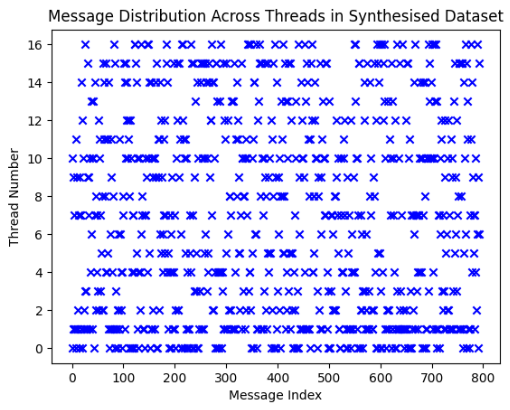


Figure 3.10: Phase 1 dataset synthesis

3.4.2 Feature selection

The primary feature used for classification was the vectorised preprocessed messages. These vectors were combined with additional features to enhance classification results. Apart from tokens, other features obtained from text include the presence of **emojis and emoticons** and the **part-of-speech tags**. Furthermore, I utilised **creator IDs** and **date and time** information as categorical and numerical features. As stated in Sections 3.2.1 and 3.2.2, CIDs were a crucial aspect for classification considering most creators only contributed to a single thread.

For SVM, these features underwent preprocessing before being concatenated into a comprehensive feature vector which was then fed into the model for classification. Conversely, for BERT and similar models, the textual, numerical and categorical data had to be split into different components before preprocessing, described in Section 3.6.4.

3.5 SVM (Phase 1)

Building and training SVM models involved using readily available pre-built classes such as the `SVC` class from scikit-learn [52]. The `SVC` class has two crucial hyperparameters, `C` and `gamma`, that required fine-tuning for optimal model performance. The hyperparameter grid search on `C` and `gamma` was conducted after all other parameters and features had been chosen.

3.5.1 Vectorisation

To allow machine learning models to train and predict from textual data, they had to first be transformed into a format that these models could process. This project explored three different vectorisation methods and implemented them using the scikit-learn library [52]: **bag of words**, **TF-IDF** and **bigrams**.

The `CountVectorizer` class, which counts the frequencies of each word in a message and represents it as a vector, was used to implement bag of words and bigram vectorisation. For bigram vectorisation, the `ngram_range` parameter was set to (2, 2) to consider only n-grams of length 2. The `TfidfVectorizer` class, which converts a collection of raw documents into a sparse matrix of TF-IDF features, was used to implement TF-IDF. Section 2.3.5 further details these methods.

3.5.2 Kernel functions

As explained in Section 2.3.1, SVMs are typically used with a linear decision boundary, but can use kernel functions to implicitly map data points into a higher-dimensional space for classification. The `kernel` argument in the `SVC` class from scikit-learn [52] allowed a kernel function to be defined and applied implicitly. This project experimented with different kernel functions to optimise the performance of SVM models.

3.6 BERT (Phase 1)

Given that BERT was pre-trained on a large and general corpus, fine-tuning the model was essential for adapting it to the requirements of this project. While Hugging Face provided a pre-trained BERT model [72] that could be readily used, fine-tuning the model on the specific task was recommended [45]. This optimised the model's performance for the particular use case.

3.6.1 Tokenisation

Similar to SVMs, textual data had to first be transformed into a representation usable by machine learning models. Hugging Face offered tokenisers for each of its pre-built models. The `BertTokenizer` was based on WordPiece [48] and returned a dictionary of tensors with keys `input_ids`, `token_type_ids` and `attention_mask` which are explained in Section 2.3.4. I implemented a function to augment the token type IDs for cases where categorical and numerical features were encoded as text, explained in Section 3.6.4.

3.6.2 Further data preparation

After tokenisation, the data was organised into a `Dataset` object containing the vectorised textual, categorical and numerical data. This was then used to create a `DataLoader` which split the data into batches suitable for input into the model.

3.6.3 Fine-tuning

Fine-tuning BERT for this task involved exploring different methods. Pre-training on the CrimeBB dataset was crucial to optimise the model for this project. Additionally, other fine-tuning methods were employed, including as pre-training on both **in-domain** and **cross-domain data**, **truncation** and **discriminative fine-tuning**.

Further pre-training

In this method of fine-tuning, the model was initially pre-trained on data outside the original dataset to increase its vocabulary, before further training on CrimeBB data. For cross-domain data, I utilised the **AG News** dataset [73]. For in-domain data, I utilised data from other boards within the CrimeBB dataset that shared similar topics, such as the Hacking Combined dataset for Hacking Tutorials as explained in Table 2.2.

Truncation

To accommodate BERT's maximum sequence length of 512 tokens and effectively process longer sequences, truncation methods were used. I implemented three different types of truncation: `head-only` which retained the first 512 tokens, `tail-only` which retained the last 512 tokens and `head-and-tail` which retained the first 130 and last 382 tokens [45]. I also chose to experiment with different methods of `head-and-tail` not covered by Sun et al. [45], such as keeping the first 382 and last 130 tokens, or keeping the first and last 256 tokens.

Discriminative fine-tuning

As explained in Section 2.3.6, discriminative fine-tuning utilises different learning rates for different layers in the model. Parameters were grouped into their corresponding layers and stored in a dictionary along with their respective learning rates. The learning rate for each layer was calculated using Equation 3.2:

$$lr_i = \frac{learning\_rate}{max(i \times \lambda, 1)} \tag{3.2}$$

Here, i ranged from 0 to 11 for BERT's 12 layers, `learning_rate` the chosen learning rate hyperparameter for the model and λ a chosen parameter. For this project, I used $\lambda = 1.6$ [49], explained in Section 2.3.2.

3.6.4 Other features

While BERT was primarily designed for processing textual data, it is possible to optimise classification results through use of categorical and numerical features. This was achieved in two main approaches: **features as text** and **extending BERT models**. These non-textual features were obtained from feature exploration and engineering explained in Sections 3.2.2 and 3.2.3. Categorical features included **one-hot encoded** features, while numerical features included **timestamps**, **offsets** and **scaled POS tags**.

As text

In this approach, categorical and numerical data were converted to strings and concatenated with textual data before tokenisation, separated by the [SEP] token [74]. For the model to differentiate between the textual and non-textual features, token type IDs were used. As BERT was originally designed to handle two sequences, token type IDs would assign either '1's or '0's to indicate sequence boundaries. Since there could be more than 2 types of features - text, categorical and numerical, I defined a new function to overwrite the token type IDs and assign different integers to separate feature types, allowing the model to discern between them and facilitating effective feature-integration.

Custom BERT

I developed a custom implementation of BERT to support the inclusion of extra hidden dimensions by expanding the hidden size of the model, seen in Listing 3.1. The input, categorical and numerical tensors from the `DataLoader` were fed into the model separately, seen in Listing B.1.

Modular BERT

The `BertWithTabular` class from the Multimodal Transformers library [75] was utilised to concatenate preprocessed categorical and numerical data with the textual data. A configuration object with the desired settings was defined before passing it as a parameter into the `BertWithTabular` class, allowing additional parameters to be specified. This configuration object also enabled the following parameters to be defined: `numerical_feature_dimensions`, `categorical_feature_dimensions` and `combine_feature_method`. The first two parameters specified the dimensions for the numerical and categorical feature tensors respectively.

The final parameter gave various options for processing the tensors before concatenation and passing them into the model. The `attention_on_cat_and_numerical_feats` option applied attention on each feature tensor. The other options were variations on passing data through a Multilayer Perceptron (MLP), described in Section 2.3.2, before concatenation. The numerical and categorical data could be passed into MLPs individually or together before concatenation with textual data.

3.6.5 Other models

I also experimented with other BERT-like models such as ALBERT, roBERTa and custom models, described in Section 2.3.3.

Custom BERT

In addition to adapting BERT to handle additional data types, custom classification heads were developed. Unlike the traditional singular linear layer used in BERT [43], these custom heads incorporated additional layers such as dropout and activation functions to mitigate overfitting. One modified classification head used included a **ReLU activation layer** and **dropout layer**, inserted between two **linear layers**, seen in Listing 3.1.

```python
class ClassificationHeadRelu(nn.Module):

    def __init__(self, config, num_extra_dims):
        super().__init__()
        # Increase Dimensions to accommodate
        # Categorical and Numerical features
        total_dims = config.hidden_size + num_extra_dims

        # Fully-Connected Layers
        self.linear = nn.Linear(total_dims, total_dims)
        self.proj = nn.Linear(total_dims, config.num_labels)
        classifier_dropout = (config.classifier_dropout
                              if config.classifier_dropout is not None
                              else config.hidden_dropout_prob)

        # Dropout Layer for Regularisation
        self.dropout = nn.Dropout(classifier_dropout)

    def forward(self, features):
        x = self.linear(features)
        x = torch.relu(x) # Activation Function
        x = self.dropout(x) # Dropout Layer
        x = self.proj(x)
        return x
```

Listing 3.1: Custom BERT with ReLU classification head

3.6.6 Parameter tuning

Parameter tuning for BERT was done using an optimizer to minimise loss. In the training loop, the optimiser's `zero_grad()` method was called to clear all accumulated gradients before processing each batch of inputs. Upon obtaining outputs from the model, they were used to update the model's parameters by calling the `step()` method on the optimizer.

The `Adam` optimizer was used for BERT, implemented with the built-in PyTorch [57] object `Adam`. This is explained in Section 2.3.2. When defining the optimiser, the model's parameters meant to be updated during training were passed as arguments. This was obtained by calling `parameters()` on the model unless more advanced fine-tuning was required, such as using different learning rates for different layers as described in Section 3.6.3.

3.6.7 Training

`train()` was called to set the model to training mode. The training loop proceeded through each epoch, iterating over the `DataLoader` to extract batched parameters for input into the model. The tqdm library [76] was used to create a progress bar to visualise training and testing progress.

`zero_grad()` was used to reset gradients to zero before backpropagation. The loss of each batch could be obtained directly from the model output or calculated separately, depending on model implementation. The model called `backward()` on the loss tensor to compute gradients which were then used to update the model's parameters. The logits, representing raw predictions, were extracted from the output and passed through the **SoftMax function** to obtain probabilities for each label. During training and validation, loss was accumulated per epoch to analyse the model's training performance. Listing B.1 shows an example training loop.

3.6.8 Validation and testing

The validation and testing loops resembled the training loop but were less involved. To begin, `eval()` was used to switch the model to evaluation mode. Unlike training, there was no need to update gradients as these datasets were not used to update model parameters. The most probable labels were obtained by calling the **SoftMax function** on the output logits from the model.

3.6.9 Overfitting

Overfitting in BERT, as described in Section 4.5.2, was a significant issue. While training loss decreased as expected, validation loss stagnated or increased, indicating overfitting to the training data. To mitigate this issue, several methods were explored: **early stopping**, **increasing dropout probabilities**, using **AdamW** and **layer-wise learning rates**.

Early stopping halted the training of the model before it overfitted to training data [77]. Dropout randomly deactivated some neurons in the forward pass, reducing the model's capacity to memorise irrelevant patterns [78]. Increasing the value of the configuration object's `hidden_dropout_prob` parameter increased dropout probabilities for the model. AdamW, implemented using the `AdamW` class from `torch.optim`, addressed the weight decay issue in Adam [79], ensuring weight decay did not interfere with adaptive learning rates. Layer-wise learning rates used the same method as **discriminative fine-tuning** described in Section 3.6.3. Figure A.1 visualises these results.

3.7 Extension: gradient boosting (Phase 1)

The `GradientBoostingClassifier` class from scikit-learn [52] was used to implement gradient boosting. All the preparation code such as splitting and preprocessing data before vectorisation was similar to the SVM implementation. A notable difference in implementation was the number of hyperparameters that had to be tuned for gradient boosting. SVM used two hyperparameters, while gradient boosting required a choice of the **number of estimators**, **learning rate** and **maximum depth**.

3.8 Phase 2 overview

As Phase 1 showed poor performance, a new approach was developed to use data in a more effective and efficient manner. This approach explored feasibility of classifying threads for unseen creators and handling new, unseen threads. This consisted of two components: pairwise classification and thread classification, aimed at addressing the shortcomings of Phase 1. Pairwise classification trained the models on pairs of sequences from the dataset, while thread classification dealt with each test message individually, classifying them into their respective threads. Phase 2 was consistent with Phase 1 in some sections. The preparation stage was the same, where datasets were obtained from the CrimeBB database, synthesised and split into train, validation and test sets. Figure 3.11 is a flow diagram of Phase 2, and Figure 3.12 is a graphical explanation of this approach. Pseudocode can be found in Algorithm 1.

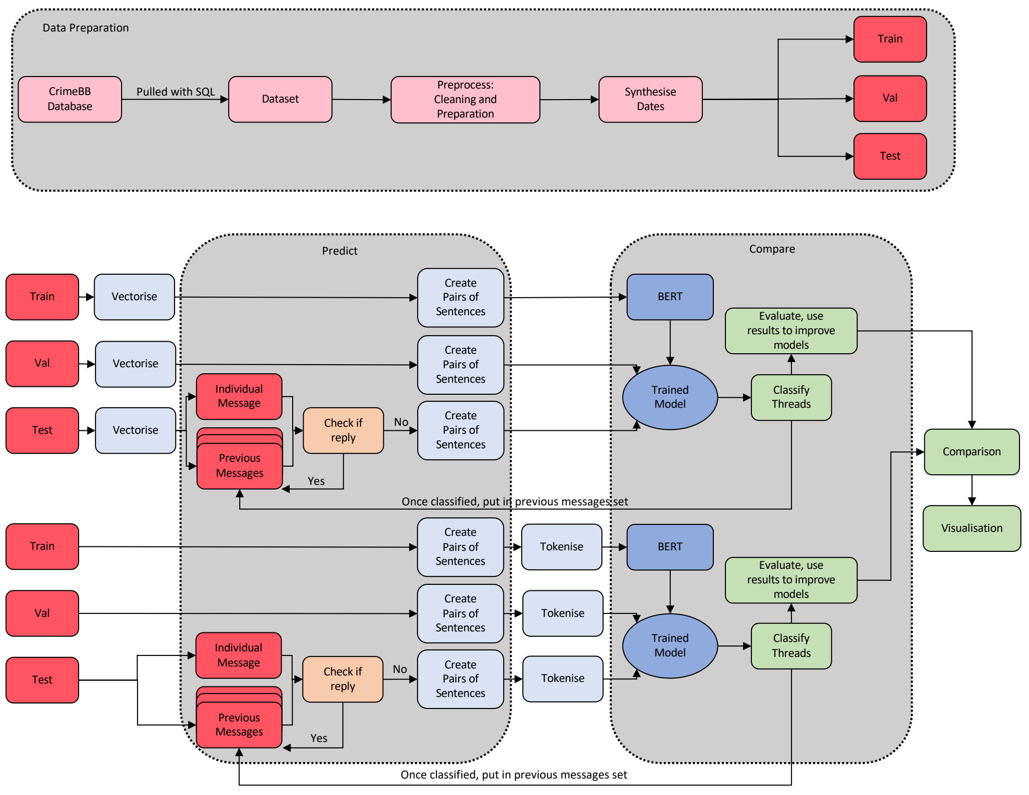


Figure 3.11: Phase 2 flow diagram

Predict and pairing

In Phase 1, the process of splitting the data into train, validation and test sets was followed directly by vectorisation and tokenisation. However, Phase 2 introduced additional steps to this workflow. For SVM, after vectorisation, individual messages from the training and validation sets were paired up and are fed into the model for pairwise classification. In contrast, for the test set, each message underwent individual testing, with messages that were tested being put into a separate set `test_seen`. The individual messages went through the predict stage, where they were tested to check if they replied to any message in `test_seen`. If so, they were added to `test_seen`, otherwise they proceeded to pairing. When pairing the test set, `test_seen` was filtered and every message in this filtered set was paired up with the single message being tested. For BERT, tokenisation only came after pairing.

Compare

In the Compare stage, paired test messages were fed into the trained models, where they were assigned threads through various classification methods. Once the message had been assigned a thread, it was added to `test_seen`. The models were evaluated and compared, similar to Phase 1.

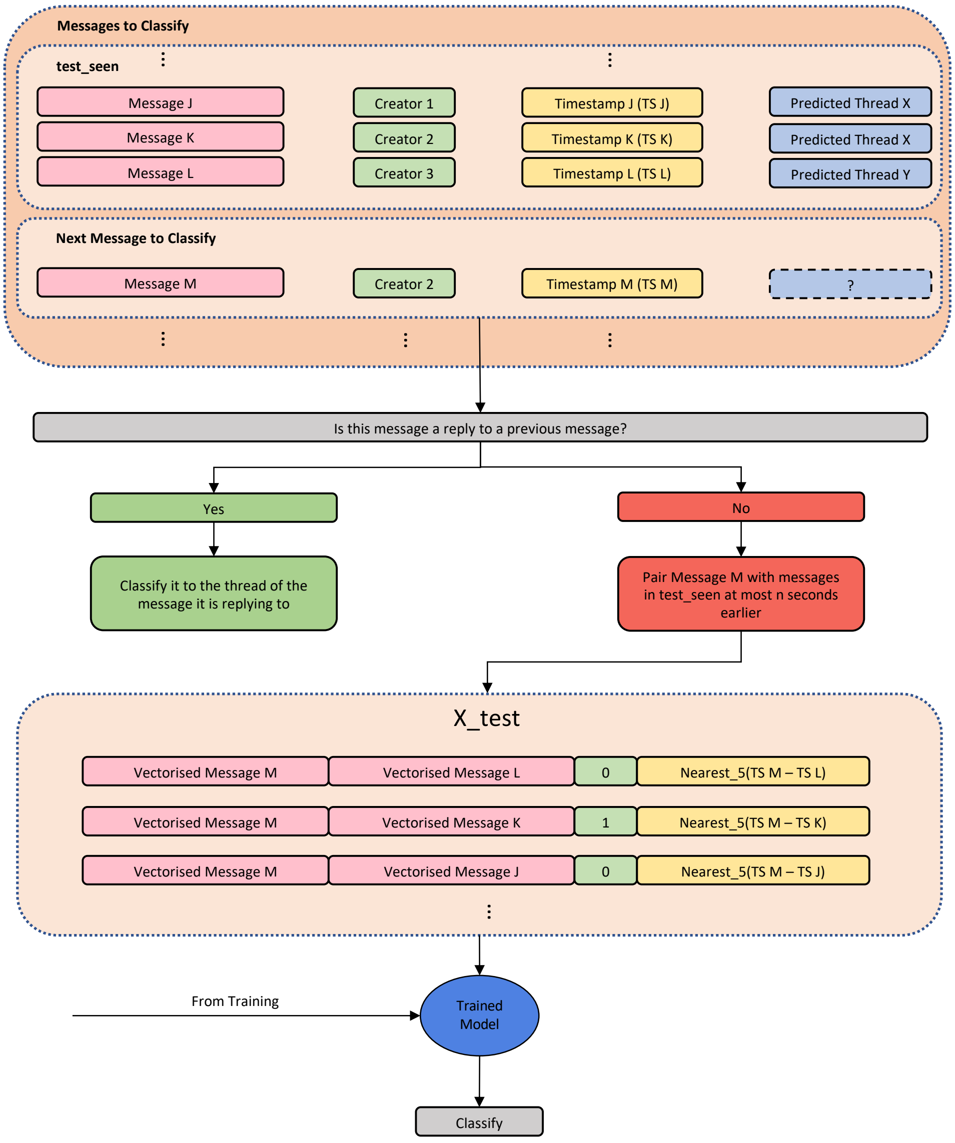


Figure 3.12: Phase 2 explanation

3.8.1 Synthesising

Instead of interleaving threads randomly, a more structured approach was adopted where the head of each thread was interleaved with the tail of the preceding thread. This facilitated smoother conversational flow resembling typical messaging platforms where threads transition naturally. Interleaving was performed at the thread-level, with messages initially sorted by `thread_id`, then by date, to maintain chronological order within each thread. For every pair of adjacent threads, `prev` and `curr`, the last $x\%$ of messages in `prev` and all messages in `curr` were interleaved, where x was a chosen parameter. This process involved probabilistically selecting the next message from either thread, with the probability of choosing from `prev` increasing every time a message from `curr` was selected. The interleaved threads became the new `prev` for the next iteration.

After interleaving, synthetic dates were generated by selecting a start timestamp and adding an offset sampled from a normal distribution. This was repeated for every message in the interleaved dataset, creating a realistic distribution of timestamps. Figure 3.13 shows the result of interleaving.

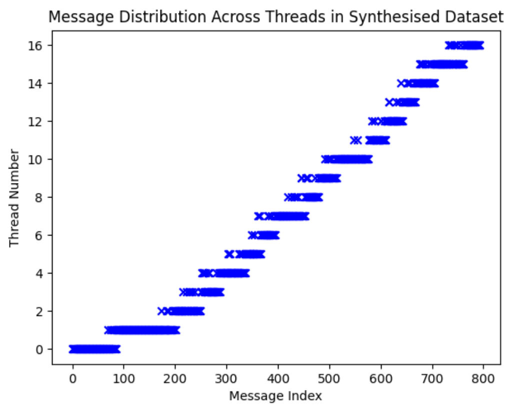


Figure 3.13: Phase 2 dataset synthesis

3.8.2 Data splitting

The dataset was split into two distinct groups with the first 60% allocated to pairwise classification and the rest to thread classification. To ensure exclusivity between the two groups, any threads present in both were restricted to the pairwise classification group. Within this group, data was further divided into train and validation sets. The first 20%was put in the train set to capture temporal relationships, with the rest split $60-40$.

Undersampling replaced oversampling used in Phase 1 to balance the dataset. The true distribution of test data was used initially but changed to $50-50$ which proved most effective. Additionally, further undersampling was applied to address the impractical size of the dataset formed from pairing sentences. For instance, the Cryptography dataset with 3,416 messages generated 24,0128 train pairs after balancing, which was impractical for training complex models like BERT. Figure 3.14 shows data splitting for Phase 2.

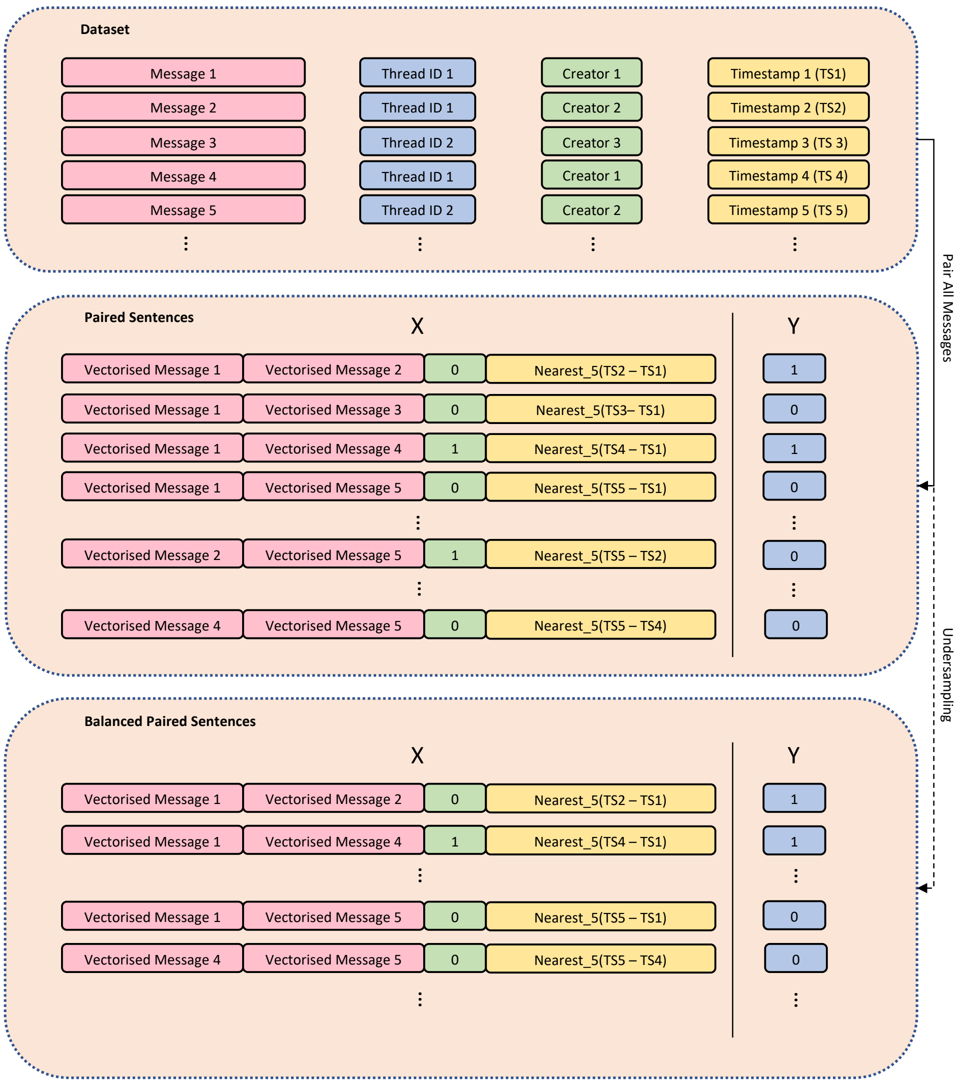


Figure 3.14: Phase 2 data splitting

3.8.3 Pairwise classification

Textual data

The textual data underwent vectorisation using the chosen vectoriser, resulting in a sparse matrix representation. Following this, data was paired, with the respective vectorised representations of each pair concatenated together. Binary y-labels indicated whether the two sentences belonged to the same thread. Data was then undersampled as explained in Section 3.8.2.

Date and time

Using the synthesised timestamps of each message, the difference `time_diff` between the timestamps of the two sequences in each pair was calculated. To reduce noise and granularity, these differences were rounded off to the nearest 5 minutes and used as a feature. This made the model less sensitive to minor differences in timestamps, allowing it to generalise better. The rounding process was implemented by first dividing `time_diff` by 300, using the `round()` function and then multiplying by 300 again. Due to the way that the dataset was synthesised, messages that were closer together were more likely to be from the same thread, aligning with the characteristics of messaging platforms.

Creator IDs

As stated in Sections 3.2.1 and 3.2.2, CIDs played a significant role in classification. Initially, raw CID values of each sequence in the pair were used, but this resulted in long training times and did not account for unseen CIDs. To address these challenges, a binary representation was adopted, where a value of '1' indicated that CIDs between the two sequences were the same, and '0' otherwise. This change not only streamlined the training process by reducing the complexity of features, but also enhanced the model's ability to deal with unseen CIDs. The resulting binary features were concatenated with other features for SVM, or put into the `DataLoader` as categorical features for BERT.

3.8.4 Thread classification

Unlike Phase 1 where messages in the test set were assessed collectively, Phase 2 adopted an individualistic approach where each message was tested sequentially. This allowed for a more granular analysis of message classification. A 'lookback' value n was used to filter messages sent within n seconds prior to the current message in the test set. This project used $n = 10800$ seconds. Thread classification involved two stages: **Predict** and **Classify**. `test_seen` was the set of seen messages within the test set, initialised with the first message in the test set.

Predict

Firstly, each message was checked to see if it was a reply to a previous message in `test_seen` by using linked messages. Should this be the case, the message was assigned to the same thread as the message it replied to. Conversely, if the message was not a reply, the message was passed into the **Classify** stage.

Classify

Here, pairs were created between singular messages and the filtered `test_seen` set which only contained messages within n seconds before the current message. These pairs were fed into the trained model for classification. From here, different methods were used to classify the message into a thread. In all classification methods, a 'decay' value was used to reduce the likelihood of a message being assigned to an existing thread, incrementing each time this happened. This was implemented as a regularisation technique, improving model adaptability and minimising the risk of erroneously classifying all messages into the same few threads.

Method 1: voting

Once predictions were obtained from the model, two methods of classification through voting were explored. The first method employed a dictionary to tally occurrences of each thread as the predicted class. When a '1', signifying a match, was predicted, the count for the corresponding thread key in the dictionary was incremented. Conversely, if a '0' was predicted, indicating no match, the count for key '0' was incremented. The message was assigned to the thread with the highest count, or a new thread if key '0' had the highest count.

The second method compared the number of mismatches with the number of matches. If the count of mismatches outweighed that of matches, the message was allocated to a new thread. Conversely if the count of matches was higher, indicating a stronger fit to existing threads, the thread with the highest count of matches was assigned as the classified thread.

Method 2: decision function

For SVMs, the `SVC` class from scikit-learn [52] provided access to the decision function through `decision_function()` which returned the signed distance of samples from the hyperplane. A positive value indicated a prediction of ‘1’, while a negative value indicated a prediction of ‘0’. In the first method utilising the decision function, the absolute maximum and minimum values from the decision function were extracted and compared. If the absolute minimum was greater, indicating a stronger negative prediction, the message was assigned to a new thread, Conversely if the absolute maximum was greater, the message was assigned to the thread associated with the message that generated this maximum value. Similar comparisons were done with the median positive and negative values which proved to work better as the maximum or minimum values may have been outliers, seen in Section 4.4a.

The second method defined a similarity threshold which moved the classification boundary from 0. I chose to use a value of 1. If the maximum value exceeded this threshold, the message was assigned to the corresponding thread. Otherwise, it was allocated to a new thread. This worked better as most of the datapoints were on the positive side of the hyperplane.

Method 3: logits

This method was only used for BERT, utilising the logit output from the model after applying the SoftMax function. Logits represented the probabilities assigned to each label, which were then normalised by the SoftMax function. The maximum values for positive and negative classes were obtained and compared, classifying the message as a new thread if the maximum for the negative class was greater. Otherwise, it was allocated to the thread providing the maximum logit value. Similar comparisons were also conducted with the median, which gave better results seen in Section 4.4b.

3.8.5 Thread merging

After the threads were classified, there were cases where too few or too many threads were formed. This was the case as Phase 2, unlike Phase 1, was not classifying messages into predefined threads. In extreme instances, the model classified every message into the same thread, or each message to its own individual thread.

In most cases, the number of predicted threads exceeded that of actual threads, possibly due to the ‘decay’ factor mentioned in Section 3.8.4. Upon inspection, many of the predicted threads could be mapped to a single actual thread, with minimal overlaps. These predicted threads were then merged based on similarity of contents using the **Jaccard score** of two threads, merging if it exceeded a defined threshold. Jaccard score divided the size of intersection between two threads, by the size of their union. Once completed, the threads were further merged based on their temporal proximity, simply by using floor division as they were numbered sequentially.

Chapter 4

Evaluation

4.1 Evaluation metrics

Establishing a standard evaluation system for all models was essential for facilitating a fair comparison. This involved training and evaluating on the same dataset, ensuring consistency across evaluations. Evaluation metrics commonly used for assessing model performance included accuracy, precision, recall and F1 score. Accuracy is a commonly used metric, taking the ratio of correct predictions to the total number of predictions. However, it can be misleading when classifying imbalanced datasets as it disregards the cost of errors, especially for the majority class [80].

Precision and recall

Precision is often used when the cost of false positives is significant. False positives would result in messages being erroneously classified into threads they did not belong in, leading to formation of large threads.

$$precision = \frac{\text{True Positive}}{\text{True Positive} + \text{False Positive}} \tag{4.1}$$

Recall is often used when the cost of false negatives is significant. False negatives would cause messages to be incorrectly separated from their actual threads, which led to the generation of many small threads.

$$recall = \frac{\text{True Positive}}{\text{True Positive} + \text{False Negative}} \tag{4.2}$$

Phase 1 involved multi-class classification, so metrics were averaged across all classes using macro-averaging. Macro-averaging calculated precision and recall for each class independently, before averaging across all classes, giving equal weight to each class.

F1 score

F1 score, a harmonic mean of precision and recall, was the main metric used for evaluation. Reducing both false positives and false negatives was important for this task.

$$F1 = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{4.3}$$

4.2 Clustering metrics

While the various classified threads may resemble clusters, evaluating their performance using clustering metrics yielded unsatisfactory results. The two metrics considered were **Jaccard score** and **silhouette score**. Jaccard score compares the similarity between predicted clusters and true clusters. However, Phase 2 involved creating new threads which led to having a different number of threads between predicted and true labels, thus this was unsuitable. Meanwhile, silhouette score quantifies how similar an object is to its own cluster, compared to other clusters, ranging from -1 to 1. The silhouette score obtained from true labels of the test set was **-0.065**, indicating overlapping or poorly separated clusters, rendering this metric unsuitable.

4.3 Testing framework

While evaluation metrics provide valuable insights into model performance, choosing the right model involves considering various factors beyond metrics alone. In software development, testing suites typically include unit, regression and integration tests to verify code behaviour and detect bugs, while code coverage analysis ensures all parts of the codebase are tested adequately. However, applying such rigorous testing methodologies to machine learning models is often impractical and unreliable due to its non-deterministic output [81]. Instead, I developed a machine learning testing framework with two main categories: pre-train tests and post-train tests [82]. Pre-train tests helped identify and rectify bugs early on, while post-train tests involved evaluating the trained model on various predefined scenarios to test for robustness.

4.3.1 Data augmentation

Data augmentation involved generating training or testing data by applying transformations and perturbations to the original dataset. For training, this process could increase the diversity and robustness of the training data, leading to improved model generalisation [83]. For testing, this evaluated the model's ability to perform despite variations in the data. Machine learning models are susceptible to adversarial attacks, leading to misinterpretations due to changes in relationships between words [84]. Various research has been done on adversarial training to regularise [85, 86] and make these models more robust [87]. This project tested the models for robustness, but adversarial training could be a possible extension for the future.

Implemented with the nlpaug library [60], textual augmentation methods enriched the testing process by introducing linguistic variations that simulate real-world variations, enabling a more thorough evaluation of the model's performance. In real-world data, typos or autocorrect may lead to unintended output of an erroneous word or symbol. **Character replacement** replaced a character within a word, while **spelling augmentation** changed the spelling of a word. **Keyboard augmentation** accounted for typos arising from character proximity on the keyboard, while **word deletion** randomly removed words from the content string, simulating instances where words were deleted during typing. **Synonym replacement** and **antonym replacement** replaced words in the text with their synonyms and antonyms respectively. **Swap** interchanged two words within the text, while **split** broke a word into two parts, creating new tokens and simulating instances where a space was inserted within a word.

4.3.2 Pre-train tests

These tests were conducted on the data to ensure the suitability of data for training, preventing potential issues during model development. To test data shape consistency, the dimensions inputs and labels were tested for alignment. Meanwhile, testing for label leakage prevented cases where document-label pairs from the training data were present in the testing data. For augmentation, mentioned in Section 4.3.1, the model was assessed for robustness, though augmentation could also improve performance by contributing to a more diversified training dataset [83].

4.3.3 Post-train tests

These tests were conducted on the models after they had been trained to verify the effectiveness of the training process and evaluate the robustness of the model. The distribution test ensured that the total probabilities output by the model summed to one, validating the consistency of the model's probability distribution. Invariance tests applied augmentations described in Section 4.3.1 to the testing data. Perturbed testing data ideally resulted in minimal changes to the model's output if the model was robust.

4.4 Sanity check

Using a .csv file to organise and review the results of classifications served as a valuable sanity check to ensure coherence and accuracy of the model's predictions. The results were structured by separating them into their respective threads, allowing for easy and efficient validation. Each message was represented in a (creator : message) format that enabled identification of both the message sender and the context of the message.

4.5 Results

4.5.1 SVM (Phase 1)

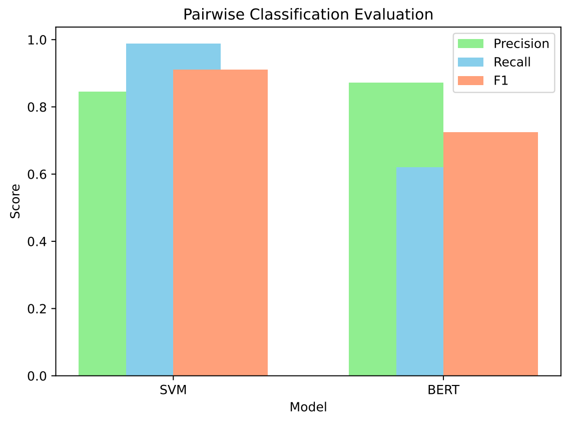
Deciding on the settings for SVM models was a crucial step before experimenting with different features. Four key settings were considered: the **vectoriser**, choice between **tokenisation or lemmatisation**, method of handling **class imbalances** and the **kernel function** to use, with results detailed in Table A.1

I then experimented with different feature representations for **date and time**, **creator IDs**, different groups of **part-of-speech tags** and **emojis and emoticons**, paired with tokens for classification, before combining them for further experiments. Results are documented in Tables A.2, A.3 and A.4.

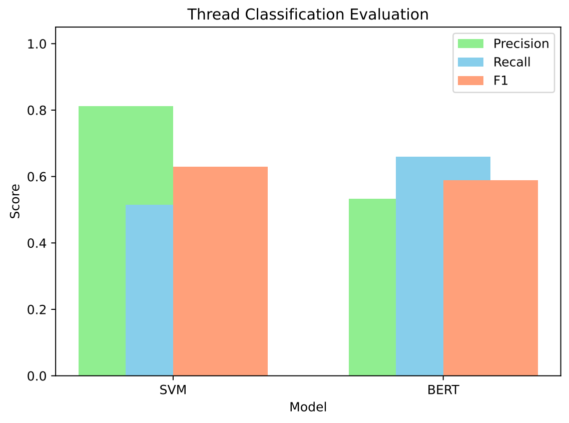
Additionally, I employed a time series train-test split to leverage the temporal nature of the data, with results presented in Table A.5. Lastly, I conducted a hyperparameter grid search over values of `C` and `gamma` to identify optimal hyperparameter values. Ultimately, the values `C` = 1 and `gamma` = 1 were selected.

4.5.2 BERT (Phase 1)

Similarly, various feature representations were explored for BERT models. A choice of model had to be made first, with results documented in Figure 4.1, though only **Modular** and **Custom BERT** models allowed non-textual data to be integrated. Moreover, the truncation method for sequences longer than 512 tokens had to be chosen, with results in Table A.6. Different combinations of features were tested to identify the most effective representation for



(a) Pairwise classification evaluation



(b) Thread classification evaluation

Figure 4.3: Phase 2 classification results

this classification task, using an initial learning rate of 5e-5, batch size of 16, and 4 epochs as recommended in the BERT paper [43]. A hyperparameter grid search was then used once the features were settled on. Results are shown in Tables A.7 and A.8.

Train and validation loss

Training the model for more epochs than recommended led to patterns observed in Figure 4.2. As expected, training loss decreased steadily over epochs, indicative of the model learning from the training data. However, validation loss exhibited a concerning trend, either plateauing or increasing after an initial decrease, indicating overfitting. Methods to mitigate this, as explained in Section 3.6.9, failed to show significant improvement, seen in Figure A.1.

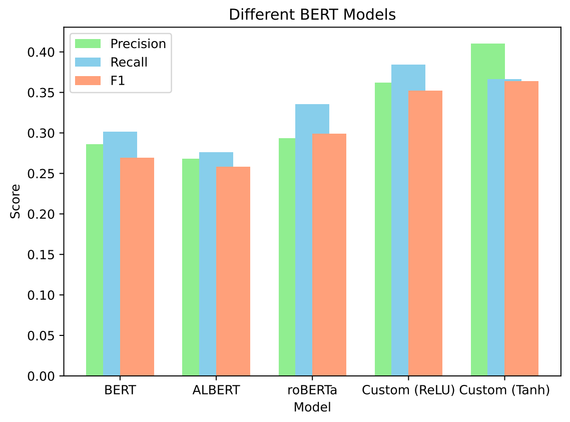


Figure 4.1: Evaluation of different BERT models

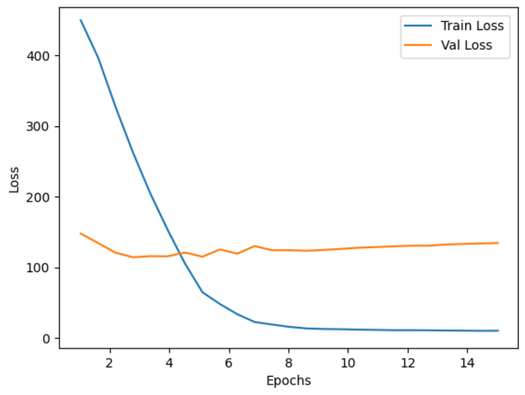
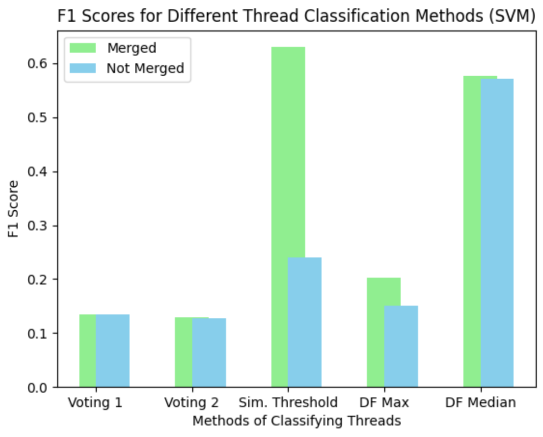


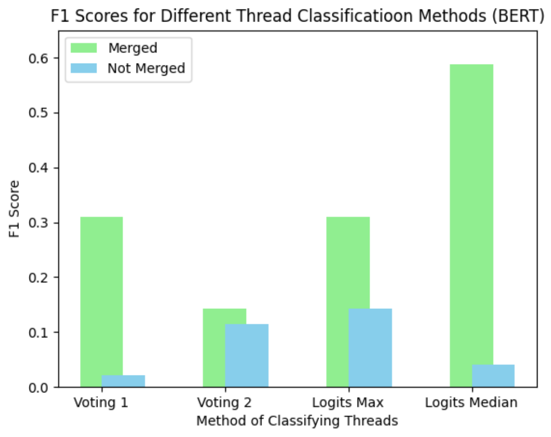
Figure 4.2: BERT training loss vs validation loss

4.5.3 Phase 2

The evaluation of Phase 2 involved assessing both **pairwise classification** and **thread classification** models, explained in Sections 3.8.3 and 3.8.4 respectively. In this phase, features were fixed, eliminating the need to test various feature representations. Results are shown in Figure 4.3. As there was no need to evaluate different representations of features like Phase 1, fewer tests were required. However, there was a need to evaluate the different methods of classification as described in Section 3.8.4. Figure 4.4 shows the results, with **similarity threshold** and **logits median** methods working best for SVM and BERT respectively.



(a) SVM (Phase 2) results



(b) BERT (Phase 2) results

Figure 4.4: Phase 2 model results

Based on metrics, SVM performed the best for Phase 2.

4.6 Comparison

The best models were chosen and compared against each other. Phase 2 provided many more datapoints to be trained by pairing messages, compared to Phase 1 which only used a dataset as large as the number of messages. In addition to BERT, other options such as ALBERT, roBERTa and custom classification heads offered versatility for different needs.

4.6.1 Metrics

In Phases 1 and 2, models were evaluated using metrics, albeit with different criteria for calculation. In Phase 1, metrics were derived from a comparison between predicted and actual thread labels. For Phase 2, metrics were calculated based on whether two messages in the test set were predicted to be in the same thread and actually belonged to the same thread. To make a proper comparison, these models had to be evaluated with the same criteria. As Phase 2 generated new threads, the number of predicted to actual threads were different, thus the Phase 1 evaluation criteria would not work. So, the Phase 2 criteria was used, with results in Figure 4.5 showing that Phase 2 SVM was the most effective model. Phase 2 SVM had the best F1 score of **0.630**, better than its Phase 1 counterpart which scored **0.337**. The next best model was Phase 2 BERT which had an F1 score of **0.588**.

[H]

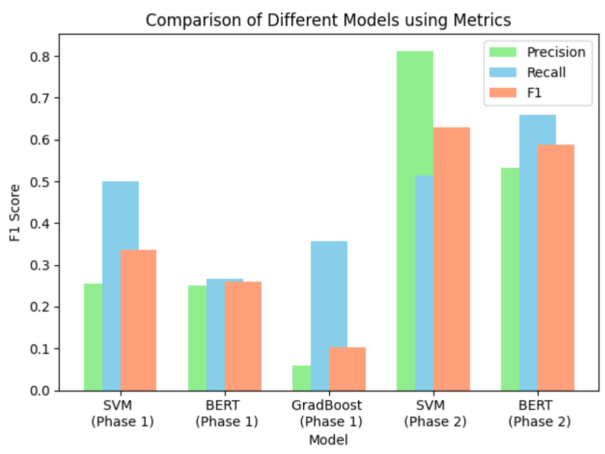


Figure 4.5: Model comparison with metrics

4.6.2 Confusion matrix

While metrics were useful, imbalanced data could lead to misleading conclusions [88]. Normalised confusion matrices showed the percentage of each class that was classified correctly in a more detailed manner. Confusion matrices in Figures 4.6, 4.7, 4.8, 4.9 and 4.10 were normalised according to the True values. From these results, Phase 2 BERT classified matches the most effectively, followed by Phase 2 SVM.

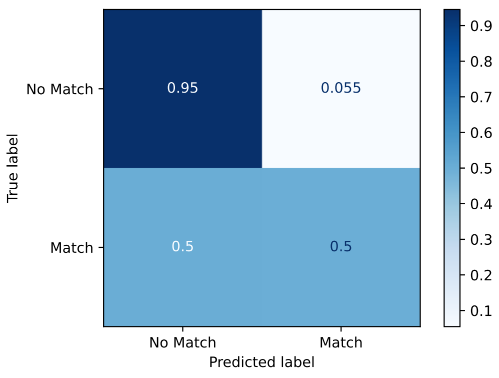


Figure 4.6: SVM (Phase 1)

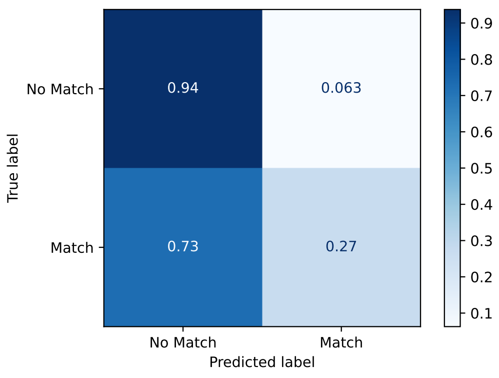


Figure 4.7: BERT (Phase 1)

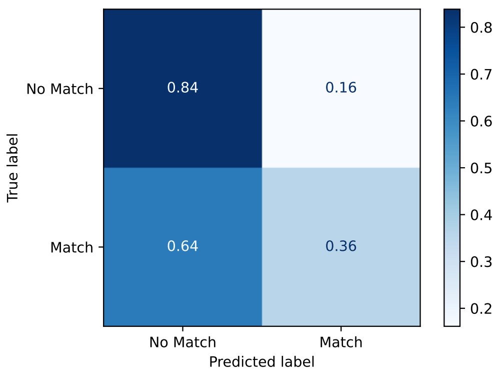


Figure 4.8: Gradient boosting (Phase 1)

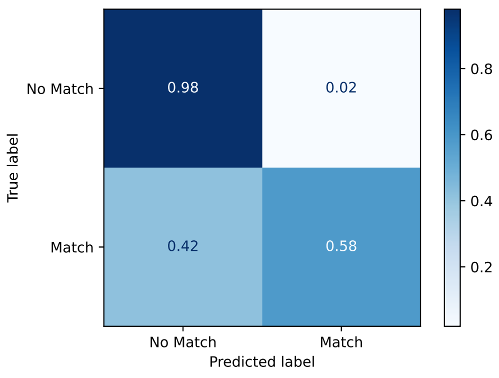


Figure 4.9: SVM (Phase 2)

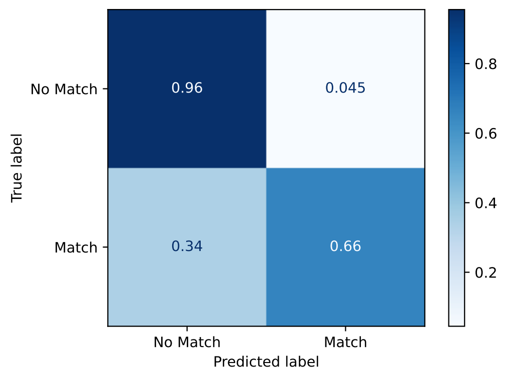
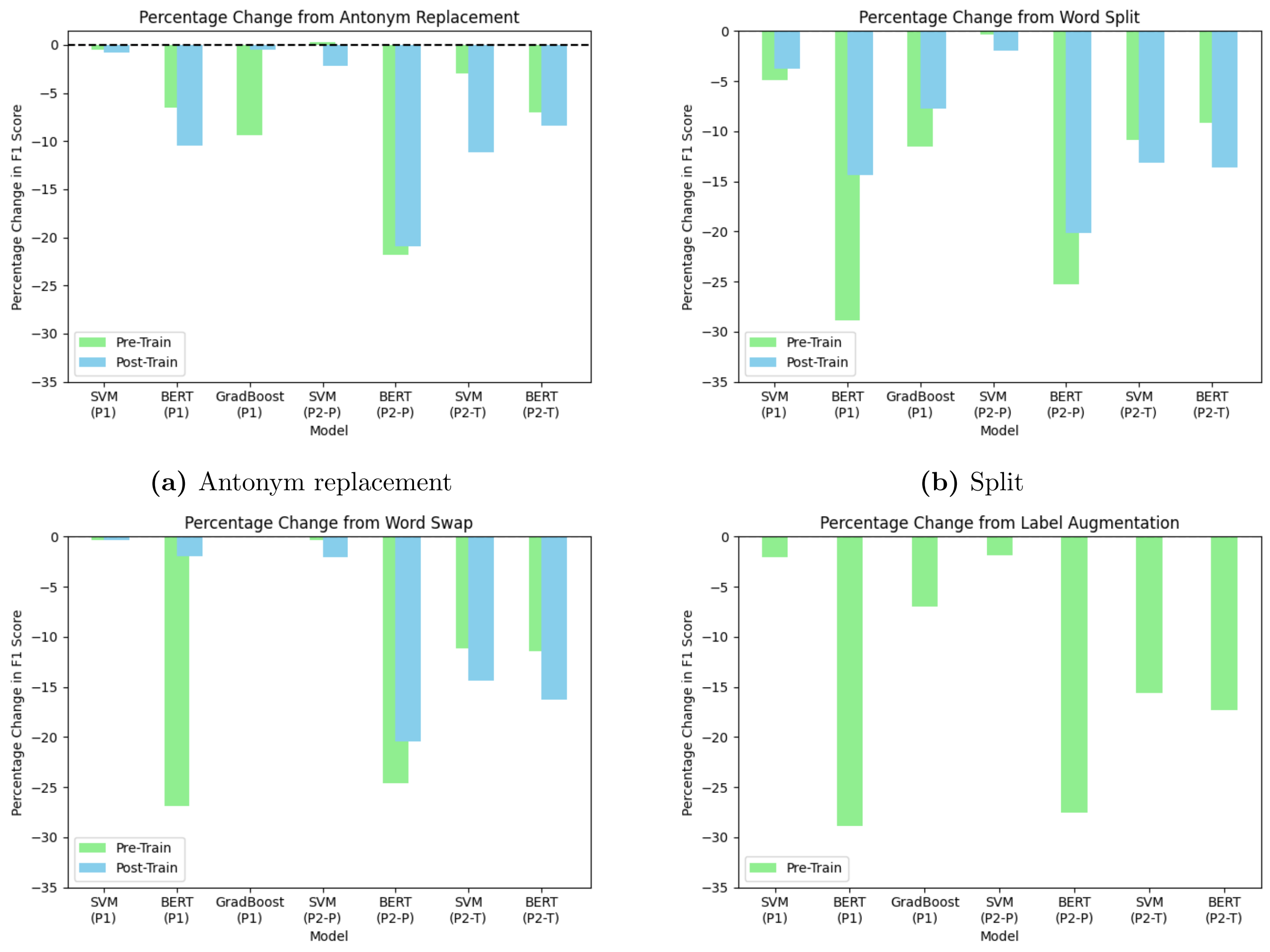


Figure 4.10: BERT (Phase 2)

4.6.3 Augmentation

Augmentation was originally meant to check for robustness, but later found to have potential performance benefits in Section 4.3.1. Figure 4.11 shows some augmentation results, with the rest in Figure A.2. P1, P2-P and P2-T represent Phase 1, Phase 2 Pairwise Classification and Phase 2 Thread Classification respectively. These results show that among the available models, SVM was the most robust among the models with respect to altered training and testing data. Moreover, perturbations such as replacing words with their antonyms even helped in classification results.



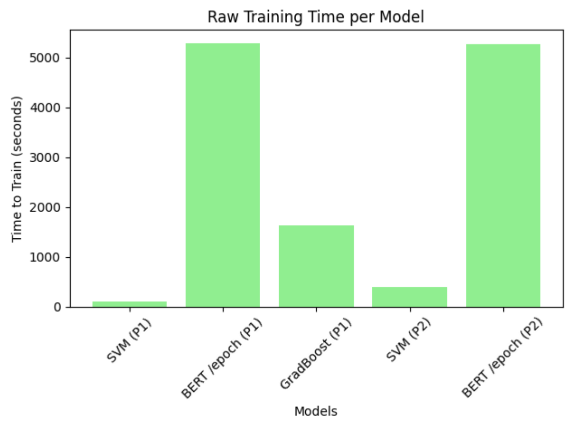
(a) Antonym replacement **(b)** Split

(c) Swap **(d)** Label augmentation

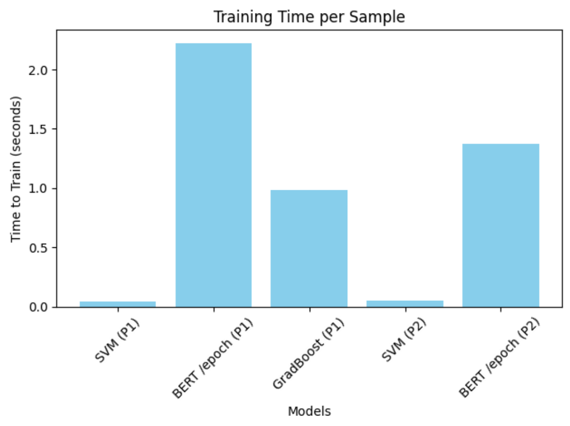
Figure 4.11: Pre-train and post-train augmentations for all models

4.6.4 Training times

Another crucial aspect considered was the training time of models. As machine learning models such as SVM and gradient boosting are smaller and simpler, they require less time to train than complex neural networks such as BERT. Models in Phase 2 were able to handle more training data due to having fewer features to manage, thus also having a lower training time per sample, seen in Figure 4.12, with Phase 1 and 2 SVMs outperforming the other models.



(a) Raw training time

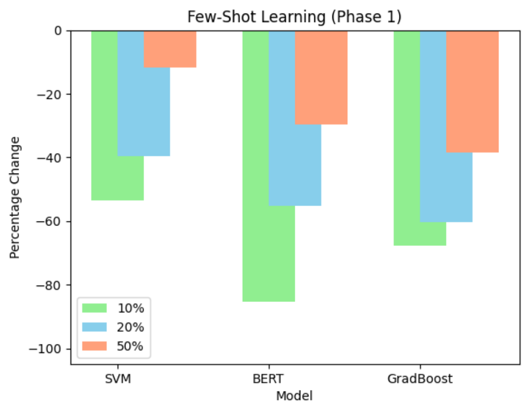


(b) Training time per sample

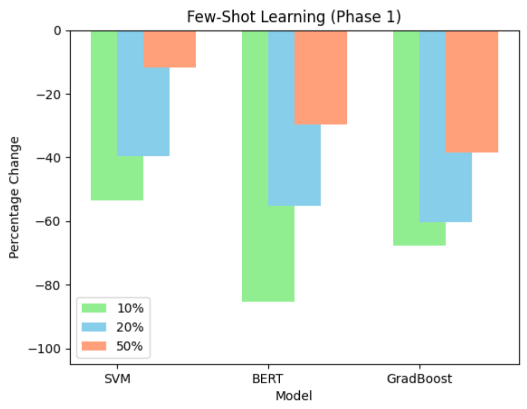
Figure 4.12: Model training times

4.6.5 Few-shot learning

Lastly, the models were compared on their ability to do few-shot learning, with results in Figure 4.13. Few-shot learning is a machine learning paradigm where a model is trained such that it is able to make predictions on a small subset of training data. As seen in Figure 4.13b, SVMs performed the best with limited training data.



(a) Few shot learning (Phase 1)



(b) Few-shot learning (Phase 2)

Figure 4.13: Percentage change in performance from few-shot learning

4.7 Application to chat data

One downside to Phase 1 models was that they required testing data to be labelled. This worked well for the CrimeBB dataset's labelled forums, but could not be used for unlabelled chat data. Meanwhile, Phase 2 was able to make classifications on unlabelled data as thread

Creator	Message
Creator A	is a vpn enough
Creator B	(Reply to A) one can not be 100 anonymous the closest to anonymity you get is tor
Creator C	the feds crawl all over that (EXPLETIVE) like ants on a three day old apple
Creator D	hello
Creator E	probably not

Table 4.1: Example Thread 1

IDs are not required in both training and classification. However, evaluation would rely on a human labelling and a sanity check on the classified threads, mentioned in Section 4.4. The typical evaluation in Section 4.6.1 required knowledge of threads, while clustering metrics were ineffective, explained in Section 4.2.

Discord data, also obtained from the Cambridge Cybercrime Centre as explained in Section 2.2, was used as input into Phase 2 models. Using Phase 2 SVM, threads occasionally contained short and unrelated messages, such as the fourth message in Table 4.1. There were also cases where threads were classified but with seemingly unrelated messages grouped together, seen in Table A.9. For cases where a conversation was ongoing between few creators, the model performed well, seen in Table A.10.

Chapter 5

Conclusions

5.1 Project reflection

There were two main questions asked at the beginning of this project.

- How do different models compare when classifying threads?
- How could the models make classifications for unseen creators or threads?

The first question was addressed in Phase 1. I carried out a detailed comparison and evaluation using both evaluation metrics and a testing framework built to test model robustness. Other aspects of the models were also evaluated, such as the training time and ability to do few-shot learning. SVM outperformed BERT in the task of classifying messages into threads, possibly due to BERT being too complex for the task, leading to overfitting and a lack of generalisation seen in Section 4.5.2. SVM worked well, especially when there was exponentially more training data than features, as seen in Phase 2.

Phase 2 answered the second question by implementing a different approach not seen elsewhere. The only features that were used were text, a binary feature indicating similarity of creators in paired messages and the time difference between the messages. New creators would not affect classification as the binary feature would be '0' whenever the two creators in the pair were different. For new messages that may be a part of an unseen thread, Phase 2 allowed new threads to be created, unlike Phase 1 which classifies only based on known threads.

The project encountered some challenges along the way, prompting me to implement innovative solutions in response. As difficulties with overfitting BERT were encountered, which was unexpected due to limited familiarity with the topic, various methods were explored to address this issue. Despite initial setbacks, the newly developed Phase 2 approach led to much better results. This shift in approach not only allowed me to overcome obstacles, but also allowed me to construct a more robust model, addressing the second question stated above.

5.2 Lessons learnt

This project was a good introduction to different machine learning models and how to fine-tune and optimise them for the task at hand. I learnt a lot about PyTorch [57] and natural language processing through this. Moreover, I learnt about testing machine learning models and best coding practices, as well as the difficulties in getting the most out of a model. Retrospectively, I would delegate more time to fine-tuning the models with different methods to improve performance, such as adversarial training mentioned in Section 4.3.1. The time required for this was underestimated, with most of the model-building time eventually spent on fine-tuning and testing new methods.

Difficulties

The main difficulties I found were issues with training time. Machine learning models take very long to train and evaluate, especially when using a large dataset. This made training and optimising models tedious and time consuming, leading to less time available for making small adjustments to the hyperparameters. Having to choose the best parameters, best optimisations and ensuring that all code worked before starting the training process was crucial so as to not waste time when an error was thrown after training and validation had completed. This necessitated better planning and highlighted the importance of reading up on literature to have the best understanding of requirements and methods before diving into the task itself.

Another difficulty I faced was with model performance. The results of the original models were not satisfactory, especially BERT, and upon analysing the issues, I had to find various different methods to alleviate the overfitting problem. Though solving this required a lot more literature to be read, I found it interesting and enjoyable to learn, implement and test the different methods.

5.3 Future work

Different models

Different models and neural networks that were not analysed in this project could be used for the task and compared. This project compared SVM, gradient boosting and BERT, but other models such as **long short-term memory (LSTM)** networks that capture long range dependencies, or a simple **multinomial Naive Bayes** model could be considered.

Supervised learning on messaging platform data

The data used for training in this task was obtained from forum threads as it was labelled with prediction ground truth, providing a quick start for training data to be used on messaging platforms. Future work on data from messaging platform requires manual labelling of the data which is an extensive process. To my knowledge, there are no readily available labelled data from messaging platforms regarding cybercrime.

Extensions to different natural languages

These models could also be extended to work on datasets with different languages. The language nuances will have to be learnt initially as structure and semantics may be different from English, but it would be an interesting horizontal improvement to see.

Bibliography

[1] National Crime Agency. Cyber crime. `https://www.nationalcrimeagency.gov.uk/what-we-do/crime-threats/cyber-crime`. Accessed: April 2024.

[2] Anna Fleck. Cybercrime expected to skyrocket in coming years. `https://www.statista.com/chart/28878/expected-cost-of-cybercrime-until-2027/#:~:text=According%20to%20estimates%20from%20Statista's,to%20%2413.82%20trillion%20by%202028.`, February 2024.

[3] Charles Griffiths. The latest 2024 cyber crime statistics. `https://aag-it.com/the-latest-cyber-crime-statistics/#:~:text=Nearly%201%20billion%20emails%20were,their%20accounts%20breached%20in%202021.`, March 2024.

[4] The Hacker News. Top industries significantly impacted by illicit telegram networks. `https://thehackernews.com/2023/08/top-industries-significantly-impacted.html`, August 2023.

[5] Jeff Burt. How cybercrims embrace messaging apps to spread malware, communicate. `https://www.theregister.com/2022/08/02/threat_groups_discord_telegram/`, August 2022.

[6] Telegram. Telegram privacy policy. `https://telegram.org/privacy?setln=fa`, note = Accessed: April 2024.

[7] Intel471. Why cybercriminals are flocking to telegram. `https://intel471.com/blog/why-cybercriminals-are-flocking-to-telegram`, August 2022.

[8] Intel471. How discord is abused for cybercrime. `https://intel471.com/blog/how-discord-is-abused-for-cybercrime`, February 2024.

[9] SOCRadar. Discord: The new playground for cybercriminals. `https://socradar.io/discord-the-new-playground-for-cybercriminals/`, May 2023.

[10] University of East London. Two thirds of european youth involved in cybercrime. `https://uel.ac.uk/about-uel/news/2022/december/two-thirds-european-youth-involved-some-form-cybercrime-online-risk-taking`, December 2022.

[11] Jonathan Lusthaus. Beneath the dark web: Excavating the layers of cybercrime's underground economy. In *2019 IEEE European Symposium on Security and Privacy Workshops*, pages 474–480, 2019. `https://ieeexplore.ieee.org/document/8802443`.

[12] DarkOwl. Understanding the difference between the surface web, deep web, and darknet. `https://www.darkowl.com/blog-content/understanding-the-difference-between-the-surface-web-deep-web-and-darknet/`, November 2022.

[13] Hannah Murphy. Telegram hits 900mn users and nears profitability as founder considers ipo. https://www.ft.com/content/8d6ceb0d-4cdb-4165-bdfa-4b95b3e07b2a, March 2024.

[14] Ash Turner. Discord users: How many people use discord? (2024). https://www.bankmycell.com/blog/number-of-discord-users/, April 2024.

[15] Liviu Trinic. Telegram becomes the new dark web for cybercrime. https://www.bitdefender.co.uk/blog/hotforsecurity/telegram-becomes-the-new-dark-web-for-cybercrime/, September 2021.

[16] Ravie Lakshmanan. Cybercriminals using telegram messenger to control toxiceye malware. https://thehackernews.com/2021/04/cybercriminals-using-telegram-messenger.html, April 2021.

[17] Intel471. How cybercriminals are using messaging apps to launch malware schemes. https://intel471.com/blog/how-cybercriminals-are-using-messaging-apps-to-launch-malware-schemes, July 2022.

[18] Richard Smedley and Neil S. Coulson. A practical guide to analysing online support forums. https://core.ac.uk/display/162667517?utm_source=pdf&utm_medium=banner&utm_campaign=pdf-decoration-v1, 2018.

[19] Giorgio Di Tizio, Gilberto Atondo Siu, Alice Hutchings, and Fabio Massacci. A graph-based stratified sampling methodology for the analysis of (underground) forums. *IEEE Transactions on Information Forensics and Security*, 18:5473–5483, 2023. http://dx.doi.org/10.1109/TIFS.2023.3304424.

[20] Are Wold. Multi-threaded conversations in instant messaging clients. https://www.duo.uio.no/bitstream/handle/10852/9666/3/Wold.pdf, May 2007.

[21] Micha Elsner and Eugene Charniak. Disentangling chat. *Computational Linguistics*, 36(3):389–409, September 2010. https://aclanthology.org/J10-3004.

[22] Jacki O'Neill and David Martin. Text chat in action. In *Proceedings of the 2003 ACM International Conference on Supporting Group Work*, GROUP '03, pages 40–49. Association for Computing Machinery, 2003. https://doi.org/10.1145/958160.958167.

[23] Shikib Mehri and Giuseppe Carenini. Chat disentanglement: Identifying semantic reply relationships with random forests and recurrent neural networks. In Greg Kondrak and Taro Watanabe, editors, *Proceedings of the Eighth International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 615–623, Taipei, Taiwan, November 2017. Asian Federation of Natural Language Processing. https://aclanthology.org/I17-1062.

[24] Jyun-Yu Jiang, Francine Chen, Yan-Ying Chen, and Wei Wang. Learning to disentangle interleaved conversational threads with a Siamese hierarchical network and similarity ranking. In Marilyn Walker, Heng Ji, and Amanda Stent, editors, *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pages 1812–1822, New Orleans, Louisiana, June 2018. Association for Computational Linguistics. https://aclanthology.org/N18-1164.

[25] Farkhund Iqbal, Benjamin C. M. Fung, Mourad Debbabi, Rabia Batool, and Andrew Marrington. Wordnet-based criminal networks mining for cybercrime investigation. *IEEE Access*, 7:22740–22755, 2019. https://ieeexplore.ieee.org/abstract/document/8606047.

[26] Khan Sameera and Pinki Vishwakarma. Cybercrime: To detect suspected user's chat using text mining, 2019.

[27] Future of Tech. Ethical issues in cybersecurity. https://www.futureoftech.org/cybersecurity/4-ethical-issues-in-cybersecurity/. Accessed: April 2024.

[28] Daniel R. Thomas, Sergio Pastrana, Alice Hutchings, Richard Clayton, and Alastair R. Beresford. Ethical issues in research using datasets of illicit origin. In *Proceedings of the 2017 Internet Measurement Conference*, pages 445–462. Association for Computing Machinery, 2017. https://doi.org/10.1145/3131365.3131389.

[29] British Society of Criminology. Statement of ethics. https://www.britsoccrim.org/ethics/. Accessed: April 2024.

[30] Sergio Pastrana, Daniel R. Thomas, Alice Hutchings, and Richard Clayton. Crimebb: Enabling cybercrime research on underground forums at scale. In *Proceedings of the 2018 World Wide Web Conference*, WWW '18, pages 1845–1854, Republic and Canton of Geneva, CHE, 2018. International World Wide Web Conferences Steering Committee. https://doi.org/10.1145/3178876.3186178.

[31] José Cabrero-Holgueras and Sergio Pastrana. A methodology for large-scale identification of related accounts in underground forums. *Comput. Secur.*, 111(C), dec 2021. https://doi.org/10.1016/j.cose.2021.102489.

[32] BBC. 'bustling' web attack market closed down. https://www.bbc.co.uk/news/technology-37859674, November 2016.

[33] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J. Alex Halderman, Luca Invernizzi, Michalis Kallitsis, Deepak Kumar, Chaz Lever, Zane Ma, Joshua Mason, Damian Menscher, Chad Seaman, Nick Sullivan, Kurt Thomas, and Yi Zhou. Understanding the mirai botnet. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1093–1110, Vancouver, BC, August 2017. USENIX Association. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/antonakakis.

[34] KrebsonSecurity. Who is anna-senpai, the mirai worm author? https://krebsonsecurity.com/2017/01/who-is-anna-senpai-the-mirai-worm-author/, January 2017.

[35] KrebsonSecurity. Source code for iot botnet 'mirai' released. https://krebsonsecurity.com/2016/10/source-code-for-iot-botnet-mirai-released/, October 2016.

[36] Patrick Howell O'Neill. Inside hackforums' rebellious cybercrime empire. https://cyberscoop.com/inside-hackforums-rebellious-cybercrime-empire/, October 2016.

[37] Rob Blaauboer. Why you shouldn't use outdated (rotten) data. https://www.yenlo.com/blogs/why-you-shouldnt-use-outdated-rotten-data/. Accessed: March 2024.

[38] Michele Banko and Eric Brill. Scaling to very very large corpora for natural language disambiguation. In *Proceedings of the 39th Annual Meeting on Association for Computational Linguistics*, ACL '01, pages 26–33. Association for Computational Linguistics, 2001. https://doi.org/10.3115/1073012.1073017.

[39] Vladimir N. Vapnik. The support vector method. https://link.springer.com/chapter/10.1007/BFb0020166, 1997.

[40] IBM. What are support vector machines (svms)? https://www.ibm.com/topics/support-vector-machine, December 2023.

[41] Larry Hardesty. Explained: Neural networks. https://news.mit.edu/2017/explained-neural-networks-deep-learning-0414, April 2017.

[42] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. https://arxiv.org/abs/1412.6980, 2017.

[43] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. https://arxiv.org/abs/1810.04805, 2019.

[44] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. https://arxiv.org/abs/1706.03762, 2023.

[45] Chi Sun, Xipeng Qiu, Yige Xu, and Xuanjing Huang. How to fine-tune bert for text classification? https://arxiv.org/abs/1905.05583, 2020.

[46] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. Albert: A lite bert for self-supervised learning of language representations. https://arxiv.org/abs/1909.11942, 2020.

[47] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. https://arxiv.org/abs/1907.11692, 2019.

[48] HuggingFace. Wordpiece tokenisation. https://huggingface.co/learn/nlp-course/en/chapter6/6. Accessed: April 2024.

[49] Dou Hu, Mengyuan Zhou, Xiyang Du, Mengfei Yuan, Meizhi Jin, Lianxin Jiang, Yang Mo, and Xiaofeng Shi. Pali-nlp at semeval-2022 task 4: Discriminative fine-tuning of transformers for patronizing and condescending language detection. https://arxiv.org/abs/2203.04616, 2022.

[50] Google For Developers. Gradient boosted decision trees. https://developers.google.com/machine-learning/decision-forests/intro-to-gbdt. Accessed: May 2024.

[51] C3.ai. Gradient-boosted decision trees (gbdt). https://c3.ai/glossary/data-science/gradient-boosted-decision-trees-gbdt/#:~:text=Gradient%2Dboosted%20decision%20trees%20are%20a%20popular%20method%20for%20solving,to%20a%20sufficiently%20optimal%20solution. Accessed: May 2024.

[52] scikit-learn: Machine learning in python. https://scikit-learn.org/stable/. Accessed: October 2023.

[53] Python Software Foundation. python. https://www.python.org, 2024.

[54] pandas Team. pandas: pandas is a fast, powerful, flexible and easy to use open source data analysis and manipulation tool, built on top of the python programming language. https://pandas.pydata.org, 2024.

[55] NumPy Team. Numpy: The fundamental package for scientific computing with python. https://numpy.org, 2024.

[56] Matplotlib: Visualization with python. https://matplotlib.org, 2023.

[57] Pytorch. https://pytorch.org. Accessed: October 2023.

[58] The ai community building the future. https://huggingface.co. Accessed: October 2023.

[59] Transformers. https://huggingface.co/docs/transformers/en/index. Accessed: October 2023.

[60] nlpaug. https://pypi.org/project/nlpaug/0.0.5/, July 2019.

[61] Visual studio code: Code editing. redefined. https://code.visualstudio.com, 2024.

[62] Gitlab: Software. faster. https://about.gitlab.com, 2024.

[63] What is the agile methodology? https://www.atlassian.com/agile#:~:text=The%20Agile%20methodology%20is%20a,planning%2C%20executing%2C%20and%20evaluating. Accessed: October 2023.

[64] Google For Developers. Imbalanced data. https://developers.google.com/machine-learning/data-prep/construct/sampling-splitting/imbalanced-data. Accessed: March 2024.

[65] Explosion. spacy: Industrial-strength natural language processing in python. https://spacy.io, 2024.

[66] NLTK Project. Natural language toolkit. https://www.nltk.org, 2023.

[67] LinkedIn. How can you balance removing outliers without losing valuable information? https://www.linkedin.com/advice/0/how-can-you-balance-removing-outliers-without-v2vbf, March 2024.

[68] Tyler Barrus. pyspellchecker. https://pypi.org/project/pyspellchecker/, January 2024.

[69] Stemming and lemmatization in python. https://www.datacamp.com/tutorial/stemming-lemmatization-python, February 2023.

[70] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer. Smote: Synthetic minority over-sampling technique. *Journal of Artificial Intelligence Research*, 16:321–357, June 2002. http://dx.doi.org/10.1613/jair.953.

[71] imbalanced-learn documentation. https://imbalanced-learn.org/stable/#, 2024.

[72] Hugging Face. Bert. https://huggingface.co/docs/transformers/en/model_doc/bert. Accessed: October 2024.

[73] Xiang Zhang, Junbo Zhao, and Yann LeCun. Character-level convolutional networks for text classification. https://arxiv.org/abs/1509.01626, 2016.

[74] Chris McCormick. Combining categorical and numerical features with text in bert, June 2021.

[75] Multimodal transformers documentation. https://multimodal-toolkit.readthedocs.io/en/latest/#multimodal-transformers-documentation, 2020.

[76] tqdm. https://tqdm.github.io, 2022.

[77] Lutz Prechelt. Early stopping-but when? http://dblp.uni-trier.de/db/conf/nips/nips1996.html#Prechelt96, 1996.

[78] Jason Brownlee. Using dropout regularization in pytorch models. https://machinelearningmastery.com/using-dropout-regularization-in-pytorch-models/, April 2023.

[79] Ilya Loshchilov and Frank Hutter. Fixing weight decay regularization in adam, 2018.

[80] Leonid A. Sevastianov and Eugene Yu. Shchetinin. On methods for improving the accuracy of multi-class classification on imbalanced data. In *Information and Telecommunication Technologies and Mathematical Modeling of High-Tech Systems 2020 (ITTMM-2020)*, pages 70–82. CEUR, 2020. https://ceur-ws.org/Vol-2639/paper-06.pdf.

[81] Dusica Marijan, Arnaud Gotlieb, and Mohit Kumar Ahuja. Challenges of testing machine learning based systems. In *2019 IEEE International Conference On Artificial Intelligence Testing (AITest)*, pages 101–102, 2019. https://ieeexplore.ieee.org/document/8718214.

[82] Jeremy Jordan. Effective testing for machine learning systems. https://www.jeremyjordan.me/testing-ml/, August 2020.

[83] Steven Y. Feng, Varun Gangal, Jason Wei, Sarath Chandar, Soroush Vosoughi, Teruko Mitamura, and Eduard H. Hovy. A survey of data augmentation approaches for NLP. *CoRR*, abs/2105.03075, 2021. https://arxiv.org/abs/2105.03075.

[84] Nicolas Papernot, Patrick McDaniel, Somesh Jha, Matt Fredrikson, Z. Berkay Celik, and Ananthram Swami. The limitations of deep learning in adversarial settings. https://arxiv.org/abs/1511.07528, 2015.

[85] Takeru Miyato, Shin ichi Maeda, Masanori Koyama, and Shin Ishii. Virtual adversarial training: A regularization method for supervised and semi-supervised learning. https://arxiv.org/abs/1704.03976, 2018.

[86] Haoming Jiang, Pengcheng He, Weizhu Chen, Xiaodong Liu, Jianfeng Gao, and Tuo Zhao. Smart: Robust and efficient fine-tuning for pre-trained natural language models through principled regularized optimization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, 2020. https://arxiv.org/abs/1911.03437.

[87] Jens Hauser, Zhao Meng, Damián Pascual, and Roger Wattenhofer. Bert is robust! a case against synonym-based adversarial examples in text classification. https://arxiv.org/abs/2109.07403, 2021.

[88] Bartosz Krawczyk. Learning from imbalanced data: open challenges and future directions. `https://link.springer.com/article/10.1007/s13748-016-0094-0?TB_iframe=true&error=cookies_not_supported&code=a3e33168-782e-41e5-8585-e731754069d2`, 2016.

Appendix A

A.1 Results

A.1.1 SVM (Phase 1)

Table A.1 shows the results from testing different settings for the SVM. From these results, the **TF-IDF Vectoriser**, **Duplicate Oversampling** and **Linear** kernel were chosen for the rest of the experiements. Next, different feature representations were tested. Various representations of **Date and Time**, **Creator IDs**, **POS Tags** and **Emojis and Emoticons** were experimented with. These results are tabulated in Table A.2. Then, combinations of these chosen feature representations were used on the SVM models, experimenting with both tokenisation and lemmatisation. Results are tabulated in Tables A.3 and A.4. Abbreviations used: CID (Creator ID), Ems (Emojis and Emoticons), POS (Part-of-Speech Tags), DoW (Day of Week). A time series split was also attempted, though the results seen in Table A.5 were not as good as the standard train-test split.

A.1.2 BERT (Phase 1)

Different truncation methods were used, as explained in Section 3.6.3, to deal with sequences longer than 512 tokens. The results are shown in Table A.6. Combinations of various representations of features were experimented with, using both the Custom BERT model with a ReLU classification head, as well as the Modular BERT models. Results are shown in Tables A.7 and A.8. Despite various efforts to tackle the issue of overfitting, the results shown in Figure A.1 were still unsatisfactory.

A.1.3 Comparisons

The pre-train and post-train augmentations tested the models for robustness to variations in data. Some results were visualised in Figure 4.11, while the rest are here in Figure A.2.

A.1.4 Example Discord Thread

Table A.9 is an example of a thread classified by Phase 2 SVM with unrelated messages put into the same thread. Table A.10 is a long example of a thread classified by Phase 2 SVM on Discord data, with only two creators.

Model	Hacking Tutorials			Cryptography		
	Precision	Recall	F1	Precision	Recall	F1
Vectoriser (BoW)	0.267	0.095	0.082	0.303	0.095	0.114
Vectoriser (TF-IDF)	**0.426**	**0.349**	**0.352**	0.119	**0.373**	**0.363**
Vectoriser (Bigrams)	0.199	0.077	0.052	**0.399**	0.198	0.220
Tokenised	0.426	0.349	0.352	0.399	0.198	0.220
Lemmatised	**0.607**	**0.405**	**0.427**	**0.471**	**0.255**	**0.286**
Class Imb. (None)	**0.602**	0.398	0.436	0.375	0.195	0.217
Class Imb. (SMOTE)	0.448	0.417	0.421	0.342	0.289	0.298
Class Imb. (Dup. Oversampling)	0.495	**0.437**	**0.451**	**0.387**	**0.298**	**0.315**
Class Imb. (Stratified TTS)	0.426	0.349	0.399	0.198	0.220	0.208
Kernel (Polynomial)	0.296	0.140	0.139	0.139	0.047	0.049
Kernel (RBF)	0.399	0.229	0.236	0.177	0.088	0.094
Kernel (Sigmoid)	0.426	0.349	0.352	0.399	0.198	0.220
Kernel (Linear)	**0.577**	**0.387**	**0.404**	**0.451**	**0.222**	**0.249**

Table A.1: SVM with different settings

Features	Hacking Tutorials			Cryptography		
	Precision	Recall	F1	Precision	Recall	F1
Timestamps	0.008	0.059	0.015	0.001	0.016	0.002
Offsets	0.011	0.042	0.017	0.021	0.048	0.027
Day of Week (Int)	0.023	0.069	0.031	0.016	0.019	0.012
Day of Week (OHE)	**0.249**	**0.171**	**0.156**	**0.112**	**0.066**	**0.065**
Date (Int)	0.008	0.059	0.015	0.001	0.016	0.002
Date (OHE)	0.064	0.077	0.043	0.044	0.038	0.028
CID (Padded Lis)t	0.020	0.076	0.029	0.004	0.025	0.007
CID (OHE)	**0.821**	**0.527**	**0.597**	**0.771**	**0.479**	**0.555**
POS Tags (All)	**0.445**	0.338	0.346	0.357	0.193	0.213
POS Tags (Main)	0.426	**0.349**	**0.352**	**0.386**	**0.194**	**0.216**
Emojis	0.608	0.373	0.400	0.437	**0.229**	**0.256**
Emoticons	0.605	0.378	0.408	0.439	0.227	0.251
Emojis and Emoticons	**0.613**	0.374	0.408	**0.446**	0.228	0.255
Emojis or Emoticons	0.608	**0.384**	**0.410**	0.440	0.227	0.251

Table A.2: SVM with different feature representations

Features	Hacking Tutorials			Cryptography		
(With Text)	Precision	Recall	F1	Precision	Recall	F1
CID	**0.820**	0.574	0.641	0.733	**0.591**	0.631
CID, Ems	0.807	0.563	0.625	**0.776**	0.575	**0.632**
CID, DoW	0.783	0.578	0.636	0.725	0.563	0.612
CID, DoW, Ems	0.772	0.580	0.636	0.731	0.559	0.611
CID, DoW, POS (All)	0.784	0.580	0.637	0.732	0.558	0.609
CID, DoW, POS (Main)	0.784	0.580	0.637	0.726	0.561	0.611
CID, DoW, POS (All), Ems	0.782	**0.590**	**0.644**	0.735	0.559	0.614
CID, DoW, POS (Main), Ems	0.773	0.583	0.639	0.732	0.562	0.614

Table A.3: SVM with different combinations of features, using tokenisation

Features (With Text)	Hacking Tutorials			Cryptography		
	Precision	Recall	F1	Precision	Recall	F1
CID	0.795	**0.624**	**0.661**	0.730	**0.572**	**0.617**
CID, Ems	0.784	0.620	0.654	**0.741**	0.570	0.615
CID, DoW	0.771	0.597	0.637	0.708	0.545	0.589
CID, DoW, Ems	0.788	0.606	0.649	0.700	0.537	0.581
CID, DoW, POS (All)	0.773	0.593	0.636	0.716	0.549	0.595
CID, DoW, POS (Main)	0.772	0.597	0.638	0.707	0.548	0.592
CID, DoW, POS (All), Ems	0.791	0.609	0.650	0.706	0.544	0.589
CID, DoW, POS (Main), Ems	**0.796**	0.615	0.657	0.703	0.544	0.588

Table A.4: SVM with different combinations of features, using lemmatisation

n	Hacking Tutorials			Cryptography		
	Precision	Recall	F1	Precision	Recall	F1
2	0.808	0.593	0.654	**0.658**	0.419	0.468
3	**0.813**	0.596	0.646	0.620	0.413	0.459
4	0.771	0.647	0.675	0.632	0.420	0.469
5	0.767	**0.671**	**0.694**	0.594	0.402	0.445
6	0.762	0.648	0.677	0.604	0.418	0.464
7	0.746	0.639	0.660	0.595	**0.449**	**0.478**

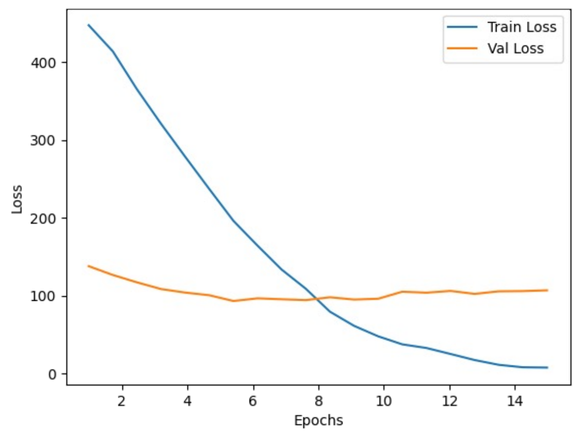
Table A.5: SVM with Time Series Train Test Split, using different values of n

Truncation Method	Precision	Recall	F1
Head-Only	**0.410**	0.366	0.364
Tail-Only	0.048	0.062	0.047
Head-and-Tail [382, 130]	0.387	**0.398**	**0.371**
Head-and-Tail [130, 382]	0.365	0.384	0.370
Head-and-Tail [256, 256]	0.324	0.341	0.332

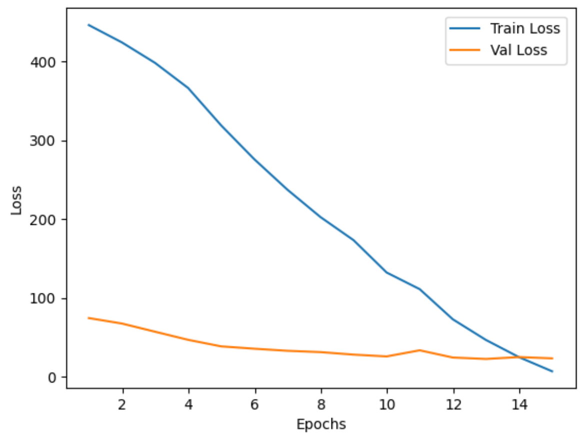
Table A.6: Different Truncation Methods with Custom ReLU Head

Features (With Text)	Cryptography		
	Precision	Recall	F1
CID	0.303	0.323	0.299
DoW	0.288	0.304	0.279
Ems	0.272	0.277	0.260
CID, DoW	**0.378**	0.362	**0.353**
CID, DoW, Ems	0.336	0.372	0.339
CID, POS (All)	0.373	0.359	0.351
CID, POS (Main)	0.344	**0.373**	0.342

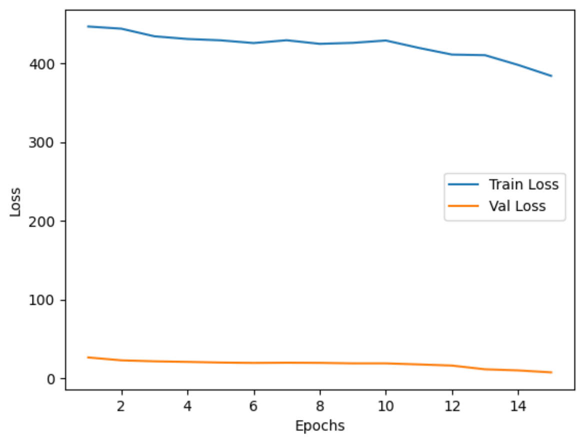
Table A.7: Custom BERT with different combinations of features



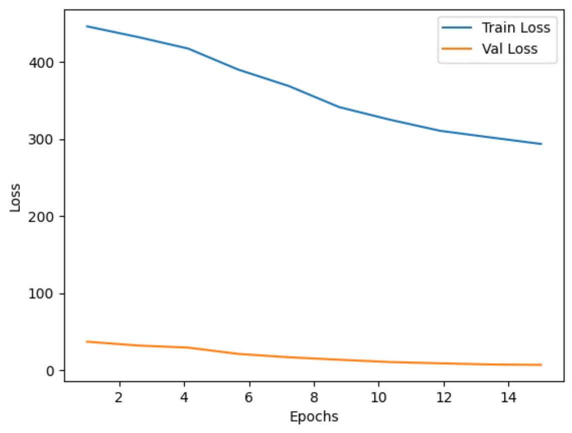
(a) Dropout with Probability 0.2



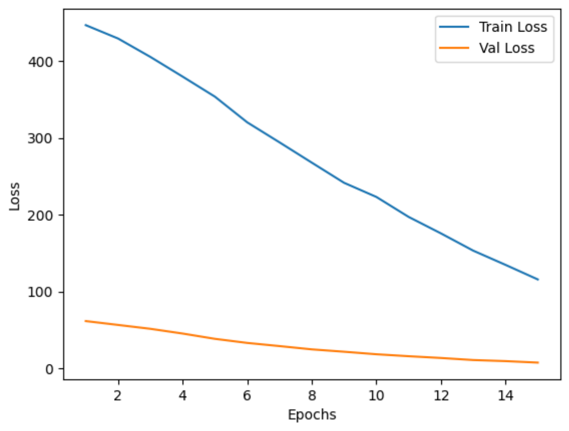
(b) Dropout with Probability 0.3



(c) Dropout with Probability 0.5

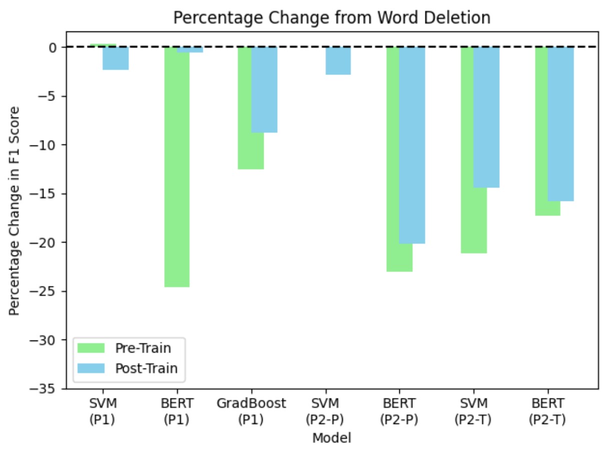


(d) AdamW Optimiser

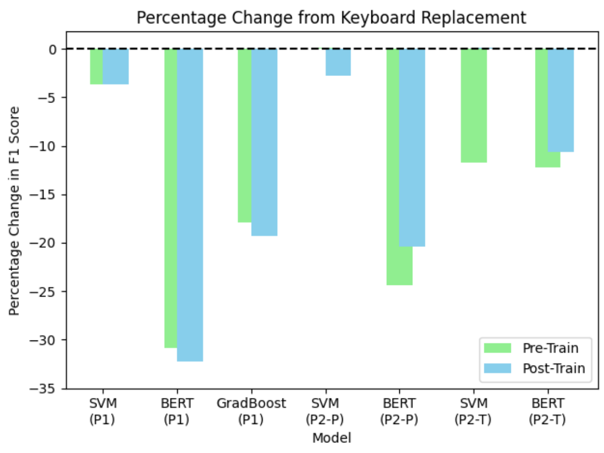


(e) Discriminative Fine-Tuning

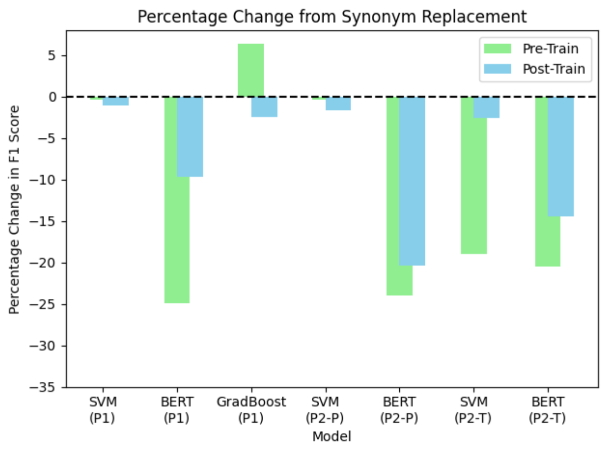
Figure A.1: Results of Applying Regularisation Methods



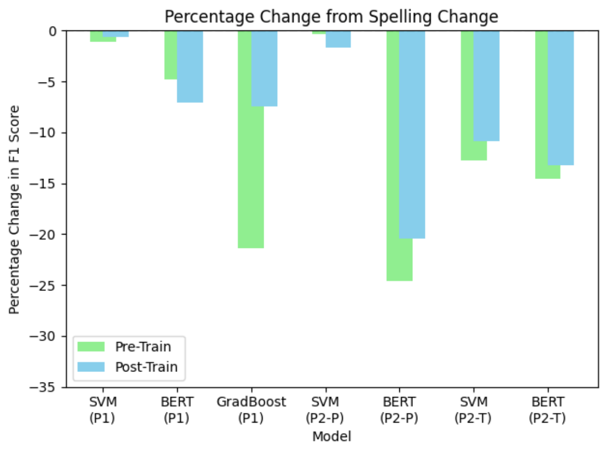
(a) Word Deletion



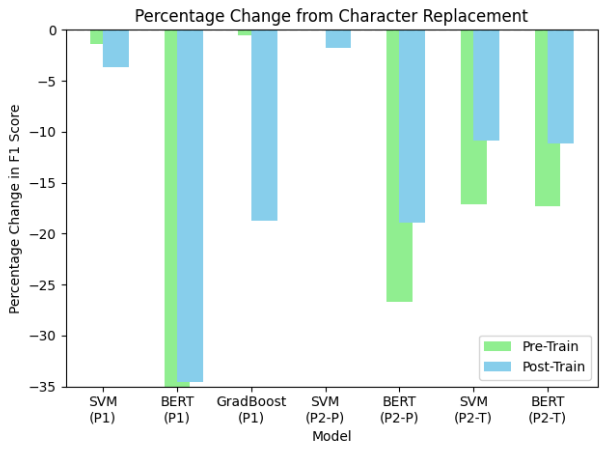
(b) Keyboard Replacement



(c) Synonym Replacement



(d) Spelling Augmentation



(e) Character Replacement

Figure A.2: Pre-Train and Post-Train Augmentations for All Models

Features (With Text)	Cryptography		
	Precision	Recall	F1
CID	**0.293**	**0.291**	**0.271**
DoW	0.273	0.277	0.260
Ems	0.284	0.220	0.224
CID, DoW	0.216	0.230	0.214
CID, DoW, Ems	0.227	0.215	0.198
CID, POS (All)	0.215	0.250	0.224
CID, POS (Main)	0.234	0.255	0.238

Table A.8: Modular BERT with different combinations of features

Creator	Message
Creator P	i was wondering is it possible to sniff answer through network analysis before the actual answer show up
Creator P	anyone here play trivia game through smartphone
Creator Q	but what do i know
Creator R	someone is rich
Creator S	is it always 8
Creator Q	i see ty
Creator T	also be mindful of what value type youre comparing x to

Table A.9: Example Thread 2

Creator	Message
Creator J	and i need help getting him to stip
Creator J	stop
Creator K	okay
Creator K	whitepages
Creator J	the phone number is (REDACTED)
Creator J	i can't afford the white pages fee
Creator J	it's 5 dollars and i can't use my dad's credit card he's kill me
Creator J	ok
Creator J	are there any other ways to find his address
Creator J	like i know he lives in (REDACTED)
Creator K	sure are
Creator K	look im sorry for you wish i could help but im not the most experienced
Creator J	do you know anyone that could help
Creator K	try this server
Creator J	he stole all my paypal email
Creator J	who's the leader of this discord
Creator J	hey guys
Creator J	i really need help
Creator K	im just a (REDACTED) neither can i
Creator K	im sure theres plenty of people that would love to help
Creator K	whatd you do
Creator J	he hacked me because i had a rare (REDACTED)
Creator J	he hacked into my email and got all of my personal email
Creator J	and he wanted to transfer my account to his email
Creator J	he knows my phone number
Creator K	well
Creator K	if you still have money
Creator J	do you know how to find someones address from their phone number
Creator J	or is there anyone in this discord that can help me
Creator J	he knows my address
Creator K	ezzz
Creator K	im out youre on your own

Table A.10: Example Thread 3

Appendix B

B.1 Code Snippets

B.1.1 BERT

Code B.1 shows a training loop for the Custom BERT model that allows categorical and numerical data to be input into the model as features.

```
def train_model(model,
                model_name,
                optimizer,
                train_loader,
                criterion,
                epoch,
                num_epochs,
                categorical=False,
                numerical=False,
                scheduler=None):
    """
    Helper function to train the model.
    This is done for a single epoch.

    Parameters:

        model (Union[BertForSequenceClassification, BertWithTabular,
        CustomBertForSequenceClassification, AlbertForSequenceClassification
    ,
        RobertaForSequenceClassification]) : The model to train.

        model_name (str) : Name of the model.
        optimizer (Union[Adam, AdamW]) : The optimizer used for optimising
        model parameters.

        train_loader (DataLoader) : The dataloader used to load data in for
        training.

        criterion (nn.CrossEntropyLoss) : The loss function to be minimised.
        epoch (int) : The current training epoch.
        num_epochs (int) : The total number of training epochs.
        categorical (bool) : Indicates whether categorical features are used
    .
        numerical (bool) : Indicates whether numerical features are used.
        scheduler (Union[None, StepLR]) : The scheduler used for training.

    Returns:

        total_loss (float) : Total training loss.
        accuracy (float) : Training accuracy.

    """
    # Set model into training mode
    model.train()
    softmax = torch.nn.Softmax(dim=0)
    total_loss = 0.0
    correct = 0
    total = 0

    if model_name == 'custom' and categorical and numerical:
        for batch in tqdm(train_loader,
                          desc=f'Epoch {epoch + 1}/{num_epochs}'):

            # Split to fit into width of dissertation
            input_batch = batch[0]
            attention_batch = batch[1]
            token_type_batch = batch[2]
            cat_batch = batch[3]
            num_batch = batch[4]
            target_batch = batch[5]

            # Zero gradients
            optimizer.zero_grad()

            # Feed parameters into model
            outputs = model(input_batch,
                            attention_mask=attention_batch,
                            token_type_ids=token_type_batch,
                            extra_data=torch.cat((cat_batch, num_batch),
                                                 dim=1),
                            labels = target_batch
                            )

            # Get loss from model output
            loss = outputs.loss

            # Backpropagation
            loss.backward()

            # Update gradients
            optimizer.step()

            # Collate loss
            total_loss += loss.item()
            logits = outputs.logits
            probs = softmax(logits)

            # Get most probable outputs
            preds = torch.argmax(probs, dim=1)
            total += target_batch.size(0)
            correct += (preds == target_batch).sum().item()

            if scheduler:
                scheduler.step()
```

Listing B.1: Training Loop for Custom BERT

Code B.2 shows the testing loop code for the Custom BERT model.

```
def test_model(model, model_name, test_loader, categorical=False, numerical=
    False):
    """
    Helper function to test the model.
    This is done for a single epoch.

    Parameters:

        model (Union[BertForSequenceClassification, BertWithTabular,
        CustomBertForSequenceClassification, AlbertForSequenceClassification,
        RobertaForSequenceClassification]) : The model to test.

        model_name (str) : Name of the model.
        test_loader (DataLoader) : The dataloader used to load data in for testing.
        categorical (bool) : Indicates whether categorical features are used.
        numerical (bool) : Indicates whether numerical features are used.

    Returns:

        all_preds (torch.Tensor) : All predictions made.
        all_probs (torch.Tensor) : All probabilities for each class.

    """
    model.eval()
    softmax = torch.nn.Softmax(dim=0)
    all_preds = []
    all_probs = []

    if model_name == 'custom' and categorical and numerical:
        with torch.no_grad():
            for input_batch, attention_batch, token_type_batch, cat_batch, num_batch, _ in tqdm(test_loader, desc='Testing'):
                outputs  = model(input_batch,
                                    attention_mask=attention_batch,
                                    token_type_ids=token_type_batch,
                                    extra_data=torch.cat((cat_batch, num_batch), dim=1)
                                    )
                logits = outputs.logits
                probs = softmax(logits)
                all_probs.append(probs)
                preds = torch.argmax(probs, dim=1)
                for pred in preds.tolist():
                    all_preds.append(pred)
    elif model_name == 'custom' and categorical:
            with torch.no_grad():
                for input_batch, attention_batch, token_type_batch, cat_batch, _ in tqdm(test_loader, desc='Testing'):
                    outputs  = model(input_batch,
                                        attention_mask=attention_batch,
                                        token_type_ids=token_type_batch,
                                        extra_data=cat_batch
                                        )
                    logits = outputs.logits
                    probs = softmax(logits)
                    all_probs.append(probs)
                    preds = torch.argmax(probs, dim=1)
                    for pred in preds.tolist():
                        all_preds.append(pred)
```

Listing B.2: Testing Loop for Custom BERT

B.1.2 Phase 2

Apart from the explanation in Section 3.8 and Figure 3.12, Algorithm 1 gives pseudocode to explain Phase 2.

Algorithm 1 Phase 2 Pseudocode

```
n ← 10800;
threads ← [ ]
for m ∈ X_test do
    if is_reply(m) then
        s ← source_message
        m.thread ← test_seen.get_thread(s)
    else
        filtered ← get_messages_in_range(m, n)
        pairs ← pair_sentences(filtered, m)
        thread ← compare(pairs)
    end if
    threads.append(thread)
end for
```

Appendix C

C.1 Project Proposal

Part II Project Proposal
Cybercrime Chat Thread Disentanglement

Ke-Ming Chong
kmc64

October 16, 2023

1 Introduction

With messaging applications such as Telegram and Discord providing smaller, more focused communities for conversations, cybercriminals are migrating towards these platforms and away from underground forums. The fragmentation of these forums onto smaller platforms allows for discussions that may not be allowed by forum administrators, such as topics including denial of service attacks, and supports variations from community standards previously set by a small group of forum administrators.

Moreover, malware creators have also moved to using Telegram as a Command-and-Control channel which adds anonymity. Forum users will often have the same username across all topics they engage in, but this may not be the case for chats. This incites cybercriminals even further to make the switch.

Within a single group chat, many different conversations may occur concurrently with one message following another but possibly having no relation whatsoever. This project aims to identify and separate different "threads" from such conversations, similar to the structure found on existing forums, accurately and label them with their respective cybercrime topics. This would help cybercrime researchers to better understand and measure topics of conversations discussed on cybercrime platforms.

2 Starting Point

I have experience with Python from the Scientific Computing course in IA, Data Science course in IB, the IB group project and personal projects. I also have experience with processing text using Python through various web scraping projects for internships.

I have learnt about supervised learning from the Machine Learning and Real-world

Data course from IA. I am taking the Natural Language Processing unit of assessment for the Michaelmas Term. The data will be obtained from the CrimeBB dataset from the Cambridge Cybercrime Centre which will require an ethics review.

3 Description

In this project, I aim to compare and contrast two main methods - Support Vector Machines and neural networks, to accurately separate threads from a conversation and compare them. A third method, gradient boosting, will be considered as an extension. Another extension will be to investigate the feasibility of feature engineering for the fine-tuning of the models.

The type of neural network to be used will be investigated in further detail in this project, with the choice of using a pre-trained model or building one from scratch also to be determined by feasibility.

Forum data will be used for this task, which contains ground truth, instead of using chat discussions directly, which would take considerable time to manually build a training set. This data will be used for training, testing and validating the models.

My findings will be recorded and presented in a dissertation.

The key concepts for this project would be the implementation and then evaluation and comparison of supervised learning methods and neural methods. Some Python libraries that may be used are NumPy and Pandas for cleaning data, scikit-learn for SVMs, Tensorflow for neural networks and XGBoost for gradient boosting.

The nature of the task requires algorithms that are suitable for multiclass classification.

- **Support Vector Machines** : The simple form of this algorithm aims to find a line that maximises the separation between a two-class dataset of points in a two-dimensional space. However, the nature of this project will require transforming the multiclass problem into binary classification in higher dimensions.
- **Neural Networks** : Consist of various layers of artificial neurons that apply an activation function to the weighted sum of its inputs plus the bias to then classify a message.
- **Gradient Boosting** : This algorithm makes use of the log-odds to make a prediction, converting these into a probability and then using this probability to classify a datapoint based on a self-defined threshold.

Some of the data structures to be used are as follows:

- **Feature Vectors** : An array of numerical values, where each element corresponds to a specific feature. In this case, it could be a specific word or suffix etc.
- **Tensors** : Multi-dimensional arrays used to represent data in neural networks. These can be scalars (0 dimensions), vectors (1 dimension), matrices (2 dimesions) and so on.
- **Trees** : Mainly for gradient boosting, consisting of nodes and branches and are used to make sequential decisions based on feature values.

4 Success Criterion

- **Core Work**
  - Built, trained and evaluated the Support Vector Machines model and neural network on the CrimeBB dataset.
- **Extensions**
  - Investigate how feature engineering may aid in making better models.
  - Investigate the feasibility of using gradient boosting for classification.

5 Work Plan

5.1 Preparation

Week 1 - 2 (3 Oct - 16 Oct)	**Work:**
	Write project proposal
	Ethics review for access to CrimeBB Dataset
	Milestones:
	Finished and submitted final proposal
	Get approval to gain access to CrimeBB dataset
6 Oct 5pm	**Phase 2 - Proposal Draft Submission**
16 Oct 5pm	**Phase 3 - Final Proposal Submission**
Week 3 - 4 (17 Oct - 30 Oct)	**Work:**
	Familiarise myself with the dataset
	Do necessary data cleaning
	Split dataset into training/test/validation sets
	Research on supervised learning
	Milestones:
	Have data ready and cleaned to be used
	Write a paragraph on supervised learning for background information

5.2 Implementation

Week 5 - 6 (31 Oct – 13 Nov)	**Work:**
	Research and create a testing framework for the models
	Research on Support Vector Machines
	Plan dissertation
	Milestones:
	Testing framework for models created
	Write a paragraph on SVMs
	Draw up a rough plan for dissertation
Week 7 - 8 (14 Nov - 27 Nov)	**Work:**
	Build, train and evaluate Support Vector Machines
	Continue planning dissertation
	Milestones:
	Support Vector Machines trained and evaluated
	Send dissertation plan to supervisor
Week 9 - 10 (28 Nov - 11 Dec)	Time off for revision and catching up
Week 11 - 12 (12 Dec - 25 Dec)	**Work:**
	Research on neural networks and pre-trained embeddings
	Improve on dissertation plan with supervisor comments
	Milestones:
	Write a paragraph on neural networks
	Decide on the type of neural network to use
	Send edited dissertation plan to supervisor

Week 13 - 14 (26 Dec - 8 Jan)	**Work:** Build and train neural network **Milestones:** Built chosen neural network
Week 15 - 16 (9 Jan - 22 Jan)	**Work:** Evaluate neural network Begin writing progress report **Milestones:** Evaluated neural network Have draft of progress report written and sent to supervisor for comments Write a few paragraphs on evaluation and comparison of both models
Week 17 - 18 (23 Jan - 5 Feb)	**Work:** Write progress report with supervisor's comments **Milestones:** Submit progress report
2 Feb 12pm	**Progress Report submission**
Week 19 - 20 (6 Feb - 19 Feb)	**Work:** Catch up on main project **Milestones:** Finish main part of project, move on to extensions

Week 21 - 22 (20 Feb - 4 Mar)	**Work:**
	Extension: Investigate feature engineering for SVM and neural network
	Milestones:
	Have tested different feature engineering methods on built and trained models
	Write a paragraph on feature engineering
Week 23 - 24 (5 Mar - 18 Mar)	**Work:**
	Extension: Investigate, build, train and evaluate gradient boosting model
	Milestones:
	Gradient boosting model has been trained and evaluated if feasible
	Write a paragraph on gradient boosting

5.3 Writing

Week 25 - 26 (19 Mar - 1 Apr)	**Work:** Begin writing dissertation (Introduction, Preparation) Continue work on extensions **Milestones:** Submit a few chapters of dissertation to supervisor Complete feasible extensions
Week 27 - 28 (2 Apr - 15 Apr)	**Work:** Write dissertation (Implementation) Update dissertation based on supervisor comments **Milestones:** Submit updated draft to supervisor for review
Week 29 - 30 (16 Apr - 29 Apr)	**Work:** Write dissertation (Evaluation, Conclusions) Update dissertation based on supervisor comments **Milestones:** Full dissertation draft complete
Week 31-32 (30 Apr - 10 May)	**Work:** Final edits to dissertation **Milestones:** Submitted dissertation
10 May 12pm	**Dissertation Deadline**
10 May 5pm	**Source Code Deadline**

6 Resource Declaration

For this project, I will use my own laptop (Macbook Air, M1 chip, 8GB RAM) for writing code, testing the models, writing the progress report and dissertation. As most of the work will be done in Python, I already have Python installed and will be using the Visual Studio Code integrated development environment, as well as Jupyter Notebook for visualisations. I accept full responsibility for this machine and I have made contingency plans to protect myself against hardware and/or software failure.

I will back up my work on a private repository on GitHub, as well as another personal laptop (Macbook Pro, Intel Core i5 chip, 8GB RAM) just in case anything happens to my main one. No data will be pushed onto the git repository to ensure that the data used is secure.