Detecting Code Words by Large-Scale Language Models

by

Nidhin Pramod

In partial fulfilment of the requirements for the degree of

MSc in

Advanced Computer Science with Big Data

A logo on a black background

Description automatically generated

Department of

Computer and Information Sciences

August, 2023

**Declaration**

This dissertation is submitted in part fulfilment of the requirements for the degree of MSc in Advanced Computer Science with Big Data of the University of Strathclyde.

I declare that this dissertation embodies the results of my own work and that it has been composed by myself.

Following normal academic conventions, I have made due acknowledgement to the work of others.

I declare that I have sought, and received, ethics approval via the Departmental Ethics Committee as appropriate to my research.

I give permission to the University of Strathclyde, Department of Computer and Information Sciences, to provide copies of the dissertation, at cost, to those who may in the future request a copy of the dissertation for private study or research.

I give permission to the University of Strathclyde, Department of Computer and Information Sciences, to place a copy of the dissertation in a publicly available archive.

Yes [ ✓ ] No [ ]

I declare that the word count for this dissertation (excluding the title page, declaration, abstract, acknowledgements, table of contents, list of illustrations, references and appendices is 15860.

I confirm that I wish this to be assessed as a Type 1 2 3 4 5 Dissertation.

Signature: A signature of a person

Description automatically generated

Date: 12-Aug-2023

Abstract

With a substantial portion of the global population having internet access, criminals have resorted to using code words to carry out their illicit activities online. It would be useful to detect such code words in real-time to aid content moderators and law enforcement agencies in mitigating crime. Detection of code words in real-time presents several challenges due to the staggering number of daily active users on several social media platforms and the ever-evolving nature of code words.

Various machine learning and deep learning models have been developed over the years for code word detection, each with its own set of complexities such as the requirement for a large dataset for training, intricate preprocessing steps and substantial hardware resources.

This paper explores the code word detection capabilities of Large Language Models (LLM) such as GPT-3, and PaLM among several others. Through several experiments, I have proved that Large Language Models can achieve state-of-the-art results at code word detection tasks, without requiring extensive training or fine-tuning of these models. LLMs significantly outperform a baseline model which was trained on the same dataset that was used to test the LLMs.

Keywords: *Code word, euphemism, Natural Language Processing, word embeddings, transformers, Large Language Models.*

Acknowledgements

I would like to express my gratitude to my supervisor Dr Dmitri Roussinov for his advice and insightful feedback that laid the foundation for this work and helped me steer my research in the right direction. I would also like to thank my parents for their constant belief in my abilities and for supporting me emotionally during times of distress.

Contents

[1 Introduction 1](#_Toc142747704)

[1.1 Problem Definition 3](#_Toc142747705)

[1.2 Research Objectives 3](#_Toc142747706)

[1.3 Applications 4](#_Toc142747707)

[1.4 Contributions 5](#_Toc142747708)

[2 Key Literature 5](#_Toc142747709)

[2.1 Preliminary Research 5](#_Toc142747710)

[2.2 Code word detection using Word2Vec word embeddings. 8](#_Toc142747711)

[2.3 Code word detection using GloVe word embeddings. 10](#_Toc142747712)

[2.4 Overview of FastText 11](#_Toc142747713)

[2.5 Transformer-based models 11](#_Toc142747714)

[2.6 Code word Detection using BERT 14](#_Toc142747715)

[2.7 Large Language Models and their key properties 15](#_Toc142747716)

[2.8 GPT-3 Model and it’s Applications. 18](#_Toc142747717)

[2.9 PaLM 2 Model 19](#_Toc142747718)

[2.10 Small LMs fine-tuned for zero-shot classification 20](#_Toc142747719)

[2.11 Summary 21](#_Toc142747720)

[3 Methodology 22](#_Toc142747721)

[3.1 Dataset Selection 22](#_Toc142747722)

[3.2 Dataset Preparation 23](#_Toc142747723)

[3.3 Experiments with GPT 32](#_Toc142747724)

[3.4 Experiments with PaLM 2 42](#_Toc142747725)

[3.5 Experiments with Zero-shot classification and small LMs 47](#_Toc142747726)

[3.6 Experiments with FastText classifier 52](#_Toc142747727)

[4 Evaluation of Results 56](#_Toc142747728)

[4.1 Evaluation Metrics Used 56](#_Toc142747729)

[4.2 Evaluation of GPT-3 Testing results 58](#_Toc142747730)

[4.3 Evaluation of PaLM 2 Testing results 62](#_Toc142747731)

[4.4 Evaluation of zero-shot classification LLMs 65](#_Toc142747732)

[4.5 Evaluation of FastText model results 68](#_Toc142747733)

[4.6 Analysis of the performance of all Models 70](#_Toc142747734)

[5 Conclusion and Recommendations 72](#_Toc142747735)

[5.1 Learnings and Experiences 72](#_Toc142747736)

[5.2 Recommendations for Future Work 73](#_Toc142747737)

[6 Bibliography 74](#_Toc142747738)

[7 Appendix 83](#_Toc142747739)

**List of Figures:**

[Figure 2‑1 Term Frequency Formula (Karabiber, 2023) 7](#_Toc142656793)

[Figure 2‑2 Inverse Document Frequency Formula (Karabiber, 2023) 7](#_Toc142656794)

[Figure 2‑3 TF-IDF Formula(Karabiber, 2023) 7](#_Toc142656795)

[Figure 2‑4 Transformer Architecture(Vaswani *et al.*, 2017) 12](#_Toc142656796)

[Figure 2‑5 Self-Supervised Code word detection Framework (Zhu *et al.*, 2021) 15](#_Toc142656797)

[Figure 2‑6 Timeline of Large Language Models developed(Zhao *et al.*, 2023) 16](#_Toc142656798)

[Figure 3‑1 BigQuery Workspace to run SQL queries 25](#_Toc142656799)

[Figure 3‑2 - Code Screenshot- BigQuery API call 26](#_Toc142656800)

[Figure 3‑3 Code screenshot - Merge code word files 27](#_Toc142656801)

[Figure 3‑4 Code screenshot- Prepare plain text corpus 28](#_Toc142656802)

[Figure 3‑5 Data cleaning: Check for the label "Yes" or "No" 29](#_Toc142656803)

[Figure 3‑6 Data cleaning: Check if the sentence contains a code word identified by the label 30](#_Toc142656804)

[Figure 3‑7 Data Cleaning: Preparing data frame 31](#_Toc142656805)

[Figure 3‑8 EDA: First five rows. 32](#_Toc142656806)

[Figure 3‑9 Zero-Shot prompt example 34](#_Toc142656807)

[Figure 3‑10 One-shot prompt example 35](#_Toc142656808)

[Figure 3‑11 Few-shot prompt example 36](#_Toc142656809)

[Figure 3‑12 GPT responds with "No" for sentence 3, 4 37](#_Toc142656810)

[Figure 3‑13 GPT responds with “No” only for sentence 3 on re-execution 37](#_Toc142656811)

[Figure 3‑14 GPT-3 testing: set API key, prompt template 39](#_Toc142656812)

[Figure 3‑15 GPT-3 Testing : Function to invoke API 40](#_Toc142656813)

[Figure 3‑16 GPT-3 testing: Driver code 41](#_Toc142656814)

[Figure 3‑17 PaLM 2 Few-shot prompt example 43](#_Toc142656815)

[Figure 3‑18 PaLM2 API parameters 45](#_Toc142656816)

[Figure 3‑19 PaLM2: Function to call Vertex AI API 46](#_Toc142656817)

[Figure 3‑20 PaLM 2 testing : Driver code 47](#_Toc142656818)

[Figure 3‑21 Zero-shot classifier example 48](#_Toc142656819)

[Figure 3‑22 Zero-Shot classifier testing: Function to call Hugging Face API 50](#_Toc142656820)

[Figure 3‑23 : Zero-shot classifier testing: Driver code 51](#_Toc142656821)

[Figure 3‑24 FastText Preprocessing Dataset 52](#_Toc142656822)

[Figure 3‑25 FastText Model training 53](#_Toc142656823)

[Figure 3‑26 FastText predict method example 55](#_Toc142656824)

[Figure 3‑27: FastText: precision and recall 55](#_Toc142656825)

[Figure 4‑1 Formula for Precision(Shung, 2018) 56](#_Toc142656826)

[Figure 4‑2 Formula for Recall(Shung, 2018) 56](#_Toc142656827)

[Figure 4‑3 Formula for Accuracy (Harikrishnan N B, 2019) 57](#_Toc142656828)

[Figure 4‑4 Confusion Matrix Columns(Harikrishnan N B, 2019) 57](#_Toc142656829)

[Figure 4‑5 GPT Evaluation: Load files 58](#_Toc142656830)

[Figure 4‑6 GPT-3 evaluation: code for confusion matric and classification report 58](#_Toc142656831)

[Figure 4‑7 GPT-3 evaluation: confusion matrix 59](#_Toc142656832)

[Figure 4‑8 GPT Evaluation: False Positives 59](#_Toc142656833)

[Figure 4‑9 : GPT-3 evaluation: and classification report 60](#_Toc142656834)

[Figure 4‑10 PaLM 2 evaluation: Confusion Matrix 62](#_Toc142656835)

[Figure 4‑11 PaLM 2 evaluation: False Negative Cases 63](#_Toc142656836)

[Figure 4‑12 PaLM2 evaluation : Confusion matrix, classification report 64](#_Toc142656837)

[Figure 4‑13 Zero-Shot classifier evaluation: load files 66](#_Toc142656838)

[Figure 4‑14 facebook/bart-large-mnli: Confusion Matrix 66](#_Toc142656839)

[Figure 4‑15: facebook/bart-large-mnli - classification report 67](#_Toc142656840)

[Figure 4‑16 MoritzLaurer/DeBERTa-v3-base-mnli-fever-anli confusion matrix 67](#_Toc142656841)

[Figure 4‑17 : MoritzLaurer/DeBERTa-v3-base-mnli-fever-anli confusion matrix and classification report 68](#_Toc142656842)

[Figure 4‑18 FastText Convert predicted labels to numerical values 69](#_Toc142656843)

[Figure 4‑19 FastText Precision, Recall and Accuracy 69](#_Toc142656844)

**List of Tables :**

[Table 1‑1 Examples of sentences containing code words 2](#_Toc142656974)

[Table 3‑1 Drugs and the corresponding Code words 24](#_Toc142656975)

[Table 3‑2 Normal Word list for preparing plain text corpus 28](#_Toc142656976)

[Table 4‑1 GPT-3 Code word Predictions 61](#_Toc142656977)

[Table 4‑2 Evaluation of PaLM2: Code word detection accuracy 65](#_Toc142656978)

[Table 4‑3 Performance of All models- Predicting presence of code word 70](#_Toc142656979)

[Table 4‑4 Performance of Models for task of identifying code words 71](#_Toc142656980)

# Introduction

The English dictionary defines a code word as “*a word or phrase that has a special meaning, different from its normal meaning, for the people who have agreed to use it in this way*” (*Code word definition and meaning | Collins English Dictionary*, no date). Throughout history, code words have been used during communication to disguise sensitive information from everyone except the intended recipient. This was done so that if the communication were to be intercepted, then the actual meaning of the message would not be clear unless the meaning of the code word was known. In times of war, armies often used various code words and cryptographic techniques to ensure the secure transmission of war strategies. In recent times, however, with the advent of the internet, code words have been used in conducting criminal activities online. As of 2021, close to 5 bn people around the world had access to the internet (Ani Petrosyan, 2023) and around 4.6 bn of them have a social media account (S. Dixon, 2023) This has resulted in an increasing number of criminals using social media and dark web to indulge in their illicit activities and expand their network (Moore, 2022). These activities include drug trafficking, planning terrorist attacks, spreading hate speech, etc. (Tom Allard, 2020). To avoid detection from law enforcement agencies and social media content moderators, these criminals often use code words and/or euphemisms to communicate among themselves and their followers or subscribers. Although encrypting messages might sound like the obvious choice for communicating malicious information, it is only used for communication between closed groups of criminals and their associates. However, to convey their messages to a larger audience, most of whom are potential followers or prospective customers, the criminals resort to using code words on social media and darknet platforms. For example, Twitter was used by several terrorist organizations for their recruitment, and it was also popular among wildlife traffickers and drug dealers (*The Crime and Terror Threat on Social Media — ACCO*, no date), (Tassone *et al.*, 2020). In 2018 Reddit banned subreddits that were dedicated to dark-net forums where users frequently indulged in discussion and business of drugs and illegal weapons (Catalin Cimpanu, 2018). Below is an illustration of example sentences containing code words for drugs and weapons:

|  |  |
| --- | --- |
| **Sentences** | **Code word meaning** |
| all up my nose already haha, was a very very fine **crystal** | methamphetamine |
| all his **fish scale** listings are showing as out of stock. | cocaine |
| I was looking to buy some **gats** from the dark web, but the they are out of stock | pistol |

Table ‑ Examples of sentences containing code words

In Table 1‑1, the words highlighted in block characters in the sentences are the code words. Such code words are used extensively for unlawful activity and detection of such code words is essential in fighting crimes and also mitigating future crimes by curtailing the outreach of criminals. The detection of code words and euphemisms has been studied extensively in linguistics with some studies that date as far back as 1997 (Pfaff, Gibbs and Johnson, 1997). More recently, the problem of code word detection was studied as a Natural Language Processing (NLP) problem using machine learning and deep learning techniques. Natural Language Processing is a branch of computer science and artificial intelligence that helps computers to comprehend and generate human language (Foote, 2023). For many years since its inception, NLP relied on a set of complex handwritten rules and eventually became unpopular due to its ineffectiveness. However, there was a resurgence in this field with the advent of machine-learning algorithms and statistical models in the 1980s and 1990s (Foote, 2023). In recent years, NLP has made remarkable progress with the advent of transformer-based Large Language Models. A Large Language Model is a language model developed using deep neural networks that are trained on large volumes of unlabelled text data, which results in models learning billions of parameters. Parameters here refer to weights and biases learned by the model during training (Zhao *et al.*, 2023).

In this paper, I have studied the code word detection capabilities of Large Language Models (LLM).

## Problem Definition

Detection of code words poses several challenges. To begin with, Human analysts manually analysing posts on social media sites is a monumental task since Twitter alone has 368 million daily active users (*500+ Social Media Statistics You Must Know in 2023*, 2023). Understanding this challenge, social media sites have an option which allows users to report posts that violate the platform’s policies. But due to the presence of code words, the true malicious nature of such posts goes unnoticed and eventually, they don’t get reported as well. With such an approach, real-time code word detection in sentences becomes impossible.

Another challenge is the constantly evolving nature of code words. Existing tools used to detect such code words are inefficient as they rely on a known “ban list” of words, and criminals often switch to using new code words to circumvent moderation policies (Cambridge Consultants, 2019). A study showed that a machine learning-based tool called Perspective meant for the detection of toxic comments could be deceived by modifying the phrases, in other words, by the usage of code words (Hosseini *et al.*, 2017).

Several other machine learning and deep learning models have been proposed for code word detection but many of these models require huge labelled datasets for training, some of them require extensive data preprocessing and involve complex techniques for the detection and identification of code words.

## Research Objectives

The main aim of this paper is to study the code word detection and identification capabilities of large language models such as GPT-3, PaLM 2, facebook/bart-large-mnli and MoritzLaurer/DeBERTa-v3-base-mnli-fever-anli. Through this study, I want to prove that Large Language Models can be used to effectively detect code words and identify what the code word means in real-time, without having to do extensive data preprocessing and without the need for a large dataset to train a model. I want to demonstrate the in-context learning capabilities of Large Language Models, that completely eliminate the need for task-specific training of the model using task-specific datasets (Zhao *et al.*, 2023)

For testing the models, I will be using a real-world dataset that contains sentences having code words in them. Using this dataset, I first want to identify the most optimal prompting technique for each of the language models listed above, by experimenting with zero-shot, one-shot and few-shot prompts. I want to design optimal prompts following the guidelines of prompt programming (Reynolds and McDonell, 2021) and use these prompts to instruct the LLMs to detect code words in the test dataset.

Following this, I want to compare the performance of each of the models and analyse if the size of the LLM, (i.e., the number of parameters) has a direct correlation with the performance of the model.

Finally, I also want to compare the performance of each of these models with a baseline model that is trained on the same dataset.

## Applications

The findings of my research have a wide range of applications. As we will see later in subsequent chapters, GPT-3 and PaLM 2 LLMs can both detect a code word in a sentence and also identify what the code word means. This is very useful for social media content moderators and law-enforcement agencies who can use my framework to detect malicious posts containing code words for criminal substances in real time. This could then be used to mitigate criminal activity on their platforms.

Other smaller LLMs like facebook/bart-large-mnli and MoritzLaurer/DeBERTa-v3-base-mnli-fever-anli can only indicate that a sentence contains a code word. This can also be useful to flag certain posts which are suspected to contain code words for criminal substances, in real time. These flagged posts could then be evaluated through human intervention for further action. These smaller LLMs are open-source models and lightweight which could be used by researchers and smaller organizations for real-time monitoring and code word detection, without the hassle of having to invest in a huge number of resources for the collection and training of data.

## Contributions

My contribution to this research is as follows:

* Preparation of Dataset using BigQuery to extract comments from banned subreddit threads of Reddit.
* Determination of optimal prompts, testing and evaluation of the GPT-3 model.
* Determination of optimal prompts, testing and evaluation of the PaLM 2 model.
* Determination of optimal prompts, testing and evaluation of facebook/bart-large-mnli and MoritzLaurer/DeBERTa-v3-base-mnli-fever-anli models.
* Data preprocessing, training, and testing of FastText classifier.

# Key Literature

This section will contain a review of previous research that has been conducted on code word detection over the years, the current state-of-the-art and recent developments in the field of Natural language processing and Large Language Models which are the core topics of my research. It will also present an overview of key properties of Large Language Models and will also describe briefly the rationale of the FastText classifier.

## Preliminary Research

Skillicorn D (2005) proposed a technique to detect code words used by illicit groups to bypass keyword filtering tools. Keyword filtering is a mechanism that was used to scan intercepted messages for specific keywords that were malicious in nature. The study made use of word frequencies and matrix decomposition techniques to detect anomalous word usage. The rationale behind this approach was the idea that the distribution of words in sentences can be used as a metric to compare two different sentences. Consequently, it would be possible to detect sentences that deviate from expected word frequency distribution, by analysing the decomposed matrices that are generated by applying matrix decomposition techniques such as Singular Value Decomposition and Independent Component Analysis which are described in the paper (Skillicorn, 2005)

Another study also proved that word frequency information can be employed at the detection of word substitution or code words in sentences, where a word with similar word frequency is used to replace another word in the sentence (Fong, Roussinov and Skillicorn, 2008). The rationale was that if two words with similar word frequencies, but very different semantic information is used to replace one another, then it creates an anomaly in the sentence that can be detected using a set of measures such as Sentence Oddity, K-gram frequencies, Pointwise Mutual Information, and Hypernym Oddity. Some of these measures treat sentences as a Bag of Words. The Bag of Words is a text representation technique used in Natural Language Processing which represents the frequency in which a word occurs in a given sentence or document (Brownlee, 2017). Sentence Oddity measures the frequency of a Bag of words for a sentence with a target word removed concerning the frequency of the entire Bag of Words. K-Gram frequency refers to the frequencies of the ‘K’ number of words in a sentence where K is a numeric integer, for example, 1-gram, 2-gram and 3-gram frequencies can be measured. All these measures were then used to identify semantic differences between regular sentences and sentences containing substitutions, to identify code words. The frequency distribution of regular sentences was obtained from Yahoo! Web search interface (Fong, Roussinov and Skillicorn, 2008). A performance analysis performed on these measures in the study by Deshmukh, Deshmukh and Deshmukh (2014) reveals that when these measures were used in tandem with a random forest classifier, the model was effective in detecting code words. There was another technique which was extensively used to represent words for NLP problems during the early 2010s, that is TF-IDF, which stands for Term Frequency and Inverse Document Frequency. Such numerical representations of words in sentences are called word embeddings. The Term Frequency measures the number of times a term appears in a document concerning the total words in a document, while Inverse Document Frequency measures the rarity of the term in a given document. The formula for Term Frequency is as follows:

A black text on a white background

Description automatically generated

Figure ‑ Term Frequency Formula (Karabiber, 2023)

The formula for Inverse Document Frequency is as follows:

A close-up of a text

Description automatically generated

Figure ‑ Inverse Document Frequency Formula (Karabiber, 2023)

The final formula for TF-IDF word embedding technique is as follows:

A black text on a white background

Description automatically generated

Figure ‑ TF-IDF Formula(Karabiber, 2023)

Both Bag of Words and TF-IDF word embedding techniques have several disadvantages, the major one being that they do not value the order of sentences and do not take into account the semantic meaning of the word in the sentences in which they appear (*Word embeddings in NLP: A Complete Guide*, no date). Due to these reasons such word embeddings were found to be less effective for tasks where understanding the semantic meaning of words in sentences was essential. This problem was solved with the innovation of the Word2Vec embedding technique (Mikolov *et al.*, 2013).

## Code word detection using Word2Vec word embeddings.

Mikolov *et al*. (2013) proposed a revolutionary word embedding technique that not only generated word embedding vectors for words but can also understand the semantic relationship between words, allowing machines to process language more effectively than before. The paper also introduces the concept of cosine similarity to determine how similar or dissimilar any two words are. Words with high cosine similarity are said to be semantically similar to each other, in other words, they appear in sentences with similar context. This is possible by measuring the cos(angle) between vector representations of the words in a higher dimensional space. Word2Vec model is created by training an artificial neural network on unlabelled text data using one of the two algorithms – Continuous Bag of Words(CBOW) or Skip-Gram (Mikolov *et al.*, 2013). In the CBOW algorithm, the model takes the context of surrounding words as input and tries to predict the current word. On the other hand, the Skip-Gram algorithm takes as input a target word, which is the current word and generates surrounding context words (*Word embeddings in NLP: A Complete Guide*, no date). The final weights obtained from the model after the completion of training are the final word embedding vectors.

Word2Vec embeddings have been used in several studies related to code word detection. Yuan *et al*. (2018) proposed a system they called the “Cantreader” that can automatically detect and identify Dark Jargons on darknet forums. Dark Jargons here refer to code words used for illicit substances on darknet forums. Cantreader attempts to identify code words by analysing the semantics of the word’s usage in communication in darknet forums and comparing that with the semantics of the word’s usage in legitimate communication. To do this comparison, they have made use of two different text corpus. The first one is the dark corpora which consists of communication from dark forums such as SilkRoad, another is the benign corpora which is a combination of text data from Reddit (legitimate corpus) and Wikipedia(reputable corpus). Unlike the traditional technique of generating Word2Vec embeddings, the researchers of this study have developed a neural network that doubles the number of input layers and takes input from both the corpora and outputs two vectors for each word, each vector corresponding to the word’s relation with other words in the each of the two corpora. Code words are then discovered by comparing the semantic similarities. If the semantic similarity between the word in the dark corpus and the legitimate corpus is less but the for the same word, there is a high semantic similarity between the legitimate corpus and reputable corpus, then the word is said to be a code word (Yuan *et al.*, 2018). After code words are identified they then use a classifier to determine the probability of a hypernym relation(is-a) between the code word and a set of target keywords or hypernyms. This model achieved a recall of 77.2% and a precision of 91% (Yuan *et al.*, 2018). This was by far the most advanced model for code word detection during this period.

However, there were other attempts at code word detection as well that made use of Word2Vec embeddings. Hada *et al*. (2021) proposed a technique to detect code words in microblogging websites by leveraging the power of Word2Vec embeddings and the cosine similarity of the word vectors. They used an approach like the one used by (Yuan *et al.*, 2018) in their study on detecting dark jargon, wherein the code words were identified based on the difference in word usage of the words in two different corpora, the good corpus and the bad corpus. However, the problem they are trying to solve here is the nature of posts on microblogging sites. On sites like Twitter, the tweets are of very short character length and consist of informal language that frequently uses slang terms and acronyms and abbreviations. After generating the word embeddings for words in both corpora, code words were then detected using a custom algorithm that compares the similarity of the same word in two corpora and looks for differences in their word usage (Hada *et al.*, 2021). All these models require training of their models on a sufficiently large enough dataset to find out differences in word usage of the word in different corpora. Without such a dataset identifying code words in real-time will not be possible as finding semantic differences for new words that may be used will not be possible, as they may be missing in either the good or bad corpus that was used during training.

Different from the above techniques, Magu *et al*. (2018) proposed a technique to detect code words in Euphemistic Hate speech that uses word embeddings from the Word2Vec model (Mikolov *et al.*, 2013) to build a hate code network. Tweets from Twitter were used to generate these word embeddings. The hate code network consisted of words on the nodes and the edges represented the cosine distance between the words. Network analysis was then performed on this hate code network to obtain properties such as the number of edges, clustering coefficient, and average degree. Code words are then identified by using word ranks obtained from centrality measures such as Eigenvector centrality (Magu and Luo, 2018). This technique relies on hate code network analysis and requires large dataset. Another drawback is that performing such a network analysis in real-time will not be possible.

## Code word detection using GloVe word embeddings.

GloVe (Pennington, Socher and Manning, 2014) was another popular word embedding technique that was used to represent text using numeric vectors. Unlike Word2Vec (Mikolov *et al.*, 2013) which uses neural networks to generate word embeddings, GloVe uses matrix factorization technique based on global word co-occurrence data in the corpus. It captures global context of the words unlike Word2Vec’s Skip-gram architecture that only captures local context. (Pennington, Socher and Manning, 2014). GloVe word embeddings were used in detection of code words in fraud investigation in a study by Zee *et al.* (2021). This paper also uses BERT word embeddings, BERT will be explained later in section 2.6. In this study, they used GloVe (Pennington, Socher and Manning, 2014) pre-trained embedding vectors in conjunction with a Bi-directional LSTM neural network to identify code words that were out of context in their sample sentence. Pre-trained word embedding vectors are pre-computed vector representations of words that have been trained on large corpora of text. The usage of pre-trained embeddings reduces the effort required to collect data and generate word embeddings from scratch (Mwiti, 2021). A bi-directional LSTM is a neural network architecture that consists of two LSTM layers, that process the input in both forward and backward directions (Bidirectional LSTM in NLP - GeeksforGeeks ). This GloVe-based bi-directional LSTM model achieved an accuracy of 80% on code word detection tasks (van der Zee *et al.*, 2021). Although this may appear to be impressive, the code words identified were known words which were replaced in the original dataset to prepare a synthetic dataset, on which the model was trained on. With the ever-evolving nature of code words, detection of unknown code words using this technique may be ineffective.

## Overview of FastText

A team of researchers at Facebook AI proposed a technique that extents the skip-gram architecture of the Word2Vec model(Mikolov *et al.*, 2013), which takes uses subword information by representing each word as a bag of character n-grams during training of the model. For example, a 3-gram representation for the word “Whale” is as follows: “wha”,”hal”,”ale”. Each n-gram would then have a corresponding vector representation and the final vector for the word would be obtained by summing up these individual n-gam representations. This is especially useful to handle rare words or misspelled words as is the case with some code words, because in a regular word2vec model, a misspelled word may not have a word embedding since the model doesn’t know its corresponding word vector generated during training (Akdogan, 2021). FastText was a library that was developed by the Facebook AI research team which is based on the same principle (Joulin *et al.*, 2016), and it can be used for text classification and word representations. This library can train a model with a large dataset in just a few minutes, without the need for tremendous resources (Joulin *et al.*, 2016). FastText has been extensively used in various NLP tasks and one study even proved that FastText outperformed Word2Vec at a sentiment analysis task related to hotel reviews(Khomsah, Ramadhani and Wijaya, 2022). For this research, I have made use of the FastText library to train a model that is capable of classifying sentences as sentences containing code and normal sentences.

## Transformer-based models

In 2017, Vaswani *et al.* (2017) proposed their transformer architecture that revolutionized the field of Natural Language Processing and Artificial Intelligence. It is considered as the foundation model as it laid the architectural foundation for the development of sophisticated Large Language Models. The most important concept introduced in this paper is the self-attention mechanism and hence the paper is aptly titled as “Attention Is All You need” (Vaswani *et al.*, 2017). Self-attention mechanism allows the model to capture long-range dependency and relationship between words that are farther apart in the sequence, this is something that previous models that used Recurring Neural Network (RNN) and Long Short-Term Memory (LSTM) network could not achieve. Another important characteristic is that the unlike previous models, the word embeddings in Transformer models also captures positional information. This is done by adding positional encoding to the input embeddings and this helps in capturing the relative or absolute position of the token in the sequence (Vaswani *et al.*, 2017). An illustration of the transformer architecture is shown below:

A diagram of a software algorithm

Description automatically generated

Figure ‑ Transformer Architecture(Vaswani *et al.*, 2017)

The inputs are fed to the encoder layer on the left which is responsible for processing the input sequence and extract meaningful representation. On the right-hand side, we have the decoder layer which is takes in the hidden states produced by the encoder and generates and sequence of output tokens (Vaswani *et al.*, 2017).

One of the most important properties of transformer-based models is the ability to perform transfer-learning (Brownlee, 2023). It is based on the idea of re-using a base model that has been pre-trained on a large dataset, for a new niche task. The base model already contains weights and biases learned during the training on the original task. This base model is then trained again on the new task specific data by adding a few task specific layers (Brownlee, 2023). This reduces the resources and time required to train a transformer-based model from scratch.

One of the earliest models that was based on transformer architecture was BERT which stands for Bidirectional Encoder Representations from Transformers (Devlin *et al.*, 2018). As it is based on the encoders from the transformer architecture, the encoding generated by BERT is considered to be context aware. As it is a bidirectional encoder, it considers context from both the right and left sides of a word in the input sequence (Devlin *et al.*, 2018). BERT was trained using the masked language model technique which involves selecting an input token in the sequence at random and masking them and then trying to predict the masked token based on the words around it. This allows BERT to capture the relationship between words and long-range dependencies in a better way. The training was done on English Wikipedia and BookCorpus with 2.5 billion words and 800 million words respectively. This makes BERT one of the earliest Large Language Models with BERT-base having learned 110 million parameters and BERT-large having learned 340 million respectively(Devlin *et al.*, 2018). The pre-trained BERT word embeddings can be fine-tuned further for specific tasks using the transfer learning process (Devlin *et al.*, 2018)

## Code word Detection using BERT

BERT has been employed in various Natural Language Processing tasks including code word and euphemism detection. One such research which I previously described in section 2.3, also used BERT based model for comparison with GloVe based BiLSTM model (van der Zee *et al.*, 2021). This study used pre-trained BERT base uncased embeddings and fine-tuned it on synthetic dataset that was prepared by substituting nouns with code words in the ENRON (CALO Project) dataset (van der Zee *et al.*, 2021). They reported that the BERT based model achieved an accuracy of 90% which is 10% more than the BiLSTM model used in the same study (van der Zee *et al.*, 2021). Another study reported surprising observations in which BERT did not perform as well as expected in comparison to the baseline model used in study that used word2vec embedding (Dessì, Recupero and Sack, 2021). This is because of the context aware nature of BERT embeddings which also includes positional encoding, due to which the representation for a word maybe different based on the order in which the word appears in a sentence. Due to this BERT embeddings cannot be treated in the same way as Word2Vec embeddings for word comparisons. The authors have suggested that different embedding for the same word could have been the cause for their surprising results (Dessì, Recupero and Sack, 2021).

The current **state-of-the-art** technique for code word and euphemism detection was proposed by Zhu *et al.* (2021). Unlike many of the papers previously described, this study uses an unsupervised training approach to first detect a set of candidate euphemisms for a given set of target key words and then uses self-supervised training approach to identify what each of those euphemistic word refers to. In the context of this paper, code words for illicit substances such as drugs, weapons etc. are referred to as euphemisms. They have used a dataset that contains code words for drugs extracted from banned subreddit and darknet forums. Code word detection is done in a two-step process, first, the task is formulated as a fill-in-the-mask problem where the target keyword in each of the sentences in the dataset is masked. The model is then trained using a Masked Language Model technique that was described in the BERT paper (Devlin *et al.*, 2018). The objective of this model during training, is to try to predict the most appropriate word that can be filled in the position of the masked word. A set of candidate euphemisms are then generated by checking the MLM probability for each word in the corpus, i.e., the probability that a word in the dataset can be used to replace the masked word. Once the set of candidate euphemism words are identified, the next step is identifying what target key word each of this euphemism refers to. This is done by creating another labelled dataset from the masked target words and using self-supervised learning scheme and a coarse-to-fine-grained classification scheme (Zhu *et al.*, 2021). All the steps of this framework are demonstrated using an illustration show below:

A close-up of a computer screen

Description automatically generated

Figure ‑ Self-Supervised Code word detection Framework (Zhu *et al.*, 2021)

This model outperforms all other previously described models with a P@10 value of 50 and P@20 value of 45 for code word detection task. This is a very complex model that involves several iterations of unsupervised and self-supervised training on the different models, the results from each network are used in another, in subsequent steps, to achieve the final objective. With larger models available today, the whole process can be simplified as demonstrated through my experiments in chapter 3.

## Large Language Models and their key properties

Large Language Models (LLMs) are complex deep learning models that are usually based on the transformer architecture and have been trained on massive datasets. Consequently, they learn millions and in some cases several billions of parameters, during their training (Zhao *et al.*, 2023). The parameters here refer to the weights and biases learned by the neural network layers during the training of the models.

The BERT language model described previously has only 340 million such parameters (Devlin *et al.*, 2018). Hence this is considered to be a small LLM. There are other such smaller LLMs such as a fine-tuned version of BART (Lewis *et al.*, 2019) that was trained on the MNLI dataset (*multi\_nli · Datasets at Hugging Face*, no date) called facebook/bart-large-mnli, which has 407 million parameters. This model along with a fine-tuned version of the DeBERTa model (He *et al.*, 2020) have been used in my experiments.

There are other significantly larger models such as T5 with 11 billion parameters(Raffel *et al.*, 2019), Pythia with 12 billion parameters (Biderman *et al.*, 2023), GPT-3 with 175 billion parameters (Brown *et al.*, 2020), and PaLM with 540 billion parameters (Chowdhery *et al.*, 2022). Such language models have displayed remarkable ability to understand language and perform complex tasks such as Text Classification, Question-Answering, Summarization, Text Generation, Translation among many others (Zhao *et al.*, 2023). Below is an illustration of all the popular large languages that have been developed in recent times:

A diagram of a company

Description automatically generated

Figure ‑ Timeline of Large Language Models developed(Zhao *et al.*, 2023)

Interaction with LLMs usually happens through a set of instructions which are input to the model and the model is expected to respond to these instructions. These instructions are referred as “prompts” and they are useful to guide the behaviour of model as well as instructing a model to perform some specific tasks(Reynolds and McDonell, 2021) .

There are certain key properties that are responsible for the remarkable performance of LLMs at Natural Language Processing tasks. These properties were referred as “Emergent Abilities” and they were described in a study by Wei *et al.* (2022). Such abilities are not present in small models but are noticed only in larger models (Wei, Tay, *et al.*, 2022). The 3 most important emergent abilities are:

* In-Context learning:

This property refers to the ability of the model to learn from demonstrations of the task and produce expected output without requiring any additional task-specific training (Wei, Tay, *et al.*, 2022)

* Instruction Following:

It is the ability of a model to perform well on unseen tasks when it has been fine-tuned on a dataset containing a set of instructions for various tasks. It also improves the ability of the model to generate better outputs for tasks that have been prompted to it in human language, in other words, for zero-shot prompts (Wei *et al.*, 2021). This process is also referred to as Instruction tuning (Zhao *et al.*, 2023). Instruction tuned models were found to have better performance at tasks that were prompted to the model (Chung *et al.*, 2022). An example of such instruction-tuned models includes Flan-T5 (Chung *et al.*, 2022) which is an instruction tuned version of T5 (Raffel *et al.*, 2019), OPT-IML which is an instruction tuned version of OPT (Iyer *et al.*, 2022).

* Step-by-step Reasoning:

It is the ability of the model to solve complex problems by breaking them down into smaller problems by having multiple reasoning steps before arriving at the final answer. The model can be instructed to do so using a prompting strategy called chain-of-thought prompting (Wei, Wang, *et al.*, 2022)

In the upcoming sections I will review Large Language Models that have these emergent capabilities.

## GPT-3 Model and it’s Applications.

All the previous models described above either require large datasets to train the model or require some level of fine-tuning of pre-trained large language models, on smaller task-specific datasets. With the aim of eliminating such limitations, GPT-3 was developed, and the details were published in a paper (Brown *et al.*, 2020). The authors of GPT-3 have described it as an autoregressive model. Autoregressive models are models which use the past values or word tokens to predict the next value or word token (Mehandzhiyski, 2023).

There was increasing evidence to suggest increasing the scale of the number of parameters correlated well with improved text synthesis and other downstream NLP tasks (Kaplan *et al.*, 2020). This model tries to exploit the potential of meta-learning or in-context learning, which is defined by the authors (Brown *et al.*, 2020) as “a process where the model develops a broad set of skills and pattern recognition abilities at training time, and then uses those abilities at inference, time to rapidly adapt to or recognize the desired task”. This essentially means that using few-shot and one-shot prompting, the model is able to understand the task at hand, recognize patterns and perform the task specified in the prompt. Few-shot prompting is prompting technique in which the prompt includes more than 1 demonstration of a specific task (Reynolds and McDonell, 2021). The example always consists of the sample input and a sample output. Similarly, one-shot prompting is prompting technique in which the prompt includes exactly one example for a specific task. On the same lines, zero-shot prompting refers to the process of instructing the prompt directly to the model without providing any illustrations (Reynolds and McDonell, 2021). The model outperformed its predecessors on a wide range of tasks, as demonstrated in the original GPT-3 paper (Brown *et al.*, 2020). Furthermore, another study (Reynolds and McDonell, 2021) argues that zero-shot prompting can sometimes match and even outperform one-shot and few-shot prompting techniques. The study also introduces prompt programming concepts, which are guidelines to generate efficient prompts to achieve the best results from GPT-3 for any given task.

Utilizing the knowledge of prompt programming in conjunction with OpenAI’s commercially available GPT-3 API endpoint (*Introduction - OpenAI API*, no date), numerous studies have been carried out to explore the full potential of GPT-3 models. One such study investigated the capabilities of using GPT-3 as a good data annotator for NLP tasks (Ding *et al.*, 2022). Another study focused on GPT-3’s short-text classification capabilities (Balkus and Yan, 2022) and the study concluded that using an augmented dataset with Open AI’s classification endpoint yielded an accuracy of 76 percent, for short text classification tasks. This proves that GPT-3 can be used for NLP tasks without having to train the model on task specific dataset.

## PaLM 2 Model

A team of researchers at Google developed a Large Language Model that had learned 540 billion parameters during its training on a large text corpus, and it was called PaLM (Chowdhery *et al.*, 2022). This model showed state-of-the-art results for various kinds off few-shot prompting tasks, due it’s in-context learning capabilities. However, the task at which PaLM excelled the most was the multi-step reasoning tasks through chain-of-thought (Wei, Wang, *et al.*, 2022) prompting (Chowdhery *et al.*, 2022). This model also demonstrated the effect of increasing the scale of large language models, proving that increasing the scale of the model improved its performance at complex tasks (Chowdhery *et al.*, 2022). PaLM 2 is an improvement on the PaLM model with improved reasoning, multilingual and code generation features (Ghahramani, 2023). This model is fairly new and was launched only as recently as May 2023 (Ghahramani, 2023). I have run a series of experiments on this model in my research to test its code word detection capabilities.

## Small LMs fine-tuned for zero-shot classification

In 2019, Yin *et al.* (2019) published a study which demonstrated a technique to use pre-trained Natural Language Interface models (NLI) as ready-made zero-shot classifiers. NLI problems are problems of identifying the correct label for a pair of sentences, in which one sentence is a premise and another sentence is a hypothesis for that premise. The model then must predict whether the hypothesis for the given premise is an entailment (true), contradiction (false) or neutral. For example, consider a premise that “The man is enjoying his pasta and dessert” and a corresponding hypothesis “The main is eating”, the correct label for the hypothesis would be “entailment” (The Stanford NLI Corpus Revisited - The Stanford Natural Language Processing Group). Similar to the NLI problem the paper proposes to treat a sentence that needs to be classified as the Premise and to develop a hypothesis using each of the candidate labels (Yin, Hay and Roth, 2019). For example, consider the sentence to be classified as follows “The man is enjoying his pasta and dessert”. And a set of candidate labels to be “eating”, “sleeping”, “exercising”. The model is expected to classify the sentence into one of the three labels, and it does this by constructing hypothesis for each of the candidate labels and treating the sentence as the Premise.

Several pre-trained models were fine-tuned based on this technique and the two popular ones are:

* facebook/bart-large-mnli (facebook/bart-large-mnli · Hugging Face)
* MoritzLaurer/DeBERTa-v3-base-mnli-fever-anli (MoritzLaurer/DeBERTa-v3-base-mnli-fever-anli · Hugging Face)

These models have been briefly explained in section 2.7 and I have also used these models in my experiments conducted in the next chapter.

## Summary

Code-word detection relied primarily on identifying words that didn’t belong in the context of the sentence. Initially, methods that made use of word frequencies were used to identify such out-of-context words and later cosine similarity measures from Word2Vec and GloVe models were used for the same task. While the initial studies used supervised learning approaches, there were unsupervised and semi-supervised approaches that were used later on. With the advent of transformer-based models, large language models such as BERT were used to detect code words in a semi-supervised training manner. Whenever something revolutionary was proposed, researchers have always tried to evaluate the new systems to understand how well these systems perform for well-known tasks such as classification, question-answer, translation, etc., in comparison to their predecessors. Similarly, GPT-3 and PaLM have also been tested for a wide range of tasks such as essay writing, text summarization, programming language code generation, text classification, and data annotation to name a few (Branwen, 2020),(Ding *et al.*, 2022),(Balkus and Yan, 2022). However, the potentials of GPT-3, PaLM 2 for code word detection tasks remain underexplored, this is what I have attempted to address in my research.

# Methodology

This section will explain in detail all the processes involved in carrying out the experiments that were conducted as part of this research.

## Dataset Selection

Previous studies in the area of code word detection have relied on several sources for preparing a text corpus of sentences containing code words. Some studies such as (Fong, Roussinov and Skillicorn, 2008) and (van der Zee *et al.*, 2021) have made use of the ENRON email dataset (CALO Project, no date). This dataset contains 0.5M emails that were originally made publicly available by the Federal Energy Regulatory Commission (CALO Project, no date). Using this ENRON email dataset, the researchers prepared a custom dataset for code word detection tasks by substituting nouns in sentences with a different noun that was out of context in the original sentence.

Several other studies have relied on scrapping malicious tweets from Twitter containing code words for drugs, weapons, and racial slurs (Hada *et al.*, 2021) and (Calderón *et al.*, 2021). The benefit of this approach over the previous approach is that there is no need to artificially synthesize the dataset. The sentences obtained from tweets have real-world applications as such tweets were used by criminals and other offenders for communication.

A few other studies have parsed dark web forums such as SilkRoad, and Darkode to extract sentences with code words and euphemisms (Yuan *et al.*, 2018). Such forums are unlikely to include a lot of code words because those forums have been developed with a specific malicious intent in mind and there is no need for the users of such forums to communicate in code words. People resort to using code words when using public platforms to avoid detection and/or moderation.

Zhu *et al.,*(2021) research made use of a dataset from multiple sources such as Reddit comments from banned subreddits, and Gab social networking sites for sentences containing code words related to drugs, weapons, and sexuality.

In like manner, for this research the dataset was prepared using comments from the subreddit forums. I decided to prepare my own dataset instead of re-using the one used in research by Zhu *et al.* (2021), because their dataset contains over 1million unlabeled sentences. Manually extracting labeling them and picking sentences containing code words I need for my research is a tedious task, because it may not contain all the drug code words that I have shortlisted. I have carefully handpicked a few words that are used in context of a code word as well as a normal word.

In 2018, Reddit banned several subreddits dedicated to selling illegal products (Catalin Cimpanu, 2018). A dump of all Reddit comments from the period 2005 to 2019 was uploaded to BigQuery and is available as a public dataset on BigQuery (*BigQuery Enterprise Data Warehouse  |  Google Cloud*, no date). The name of the dataset on BigQuery is `fh-bigquery`.

BigQuery is an enterprise data warehouse solution that is part of the Google Cloud Platform. One of the main advantages of BigQuery is that running queries on large volumes of data only takes a few milliseconds and these queries can be executed using a query language similar to what would be used in a relational database (Lakshmanan Valliappa and Tigani Jordan, 2019)

## Dataset Preparation

This research makes use of a dataset prepared using the comments extracted from subreddit threads on Reddit. The dataset consists of two distinct text corpora:

* Corpus consisting of sentences with code words for drugs. I will refer to this as code word corpora in the subsequent sections of this report.
* Corpus consisting of sentences without code words. I will refer to this as plain text corpora.

While the corpus obtained from Reddit forms the dataset for testing the Large Language Models, we need a way to validate the results of testing. For this purpose, I use an intelligence report published by the US DEA (DEA Intelligence Report, 2018). This report consists of a list of known code words for various types of drugs. This report will be referred to as the truth value document.

### Preparation of the code word corpora

There are thousands of comments posted across different subreddits every day and the banned subreddits could have thousands of comments containing malicious code words. I have primarily focused on extracting comments from r/Drugs and r/DarkNetMarkets as they were banned for selling illegal drugs (Catalin Cimpanu, 2018).

To begin with, I have prepared a list of popular drugs and their known code words from the truth value document. The following table shows the list of drugs and their corresponding code words used to extract sentences containing code words:

|  |  |
| --- | --- |
| Drugs | Known Code Words |
| Cocaine: | Baby Powder,Nose Powder,Dust,Gold Dust,Coca-Cola,Coke,Flour,Fish Scale |
| Ecstasy: | Candy, Adam, Kleenex, Love Drug, Skittle |
| Fentanyl: | Apache,China Town, China White,Blue Diamond,Fenty,Snowflake,Dance Fever |
| Heroin: | A-Bomb, Birria, Black Sheep,Brown Sugar, Chiva, Flea Powder, La Tierra |
| Ketamine: | Barry Farrell,Kit Kat, Cat Food, Green K, Vitamin K |
| LSD: | Acid, Alice, White Dust,Haze,Crackers |
| Marijuana: | Grasshopper,Blue Jeans,Cookie,Ganja,Grass,Hash,Herb,Weed,Pot |
| Methamphetamine: | Hot Ice,Chalk,Chicken Feed,Crystal,Salt |
| Mescaline: | Shaman,Hikori,Cactus |
| PCP: | Amoeba,Elephant,Gorilla Biscuits,Horse Tranquilizers,Mint Leaf,Zombie |

Table ‑ Drugs and the corresponding Code words

These code words have been carefully chosen and over half of the code words in the list are commonly used words. This is to ensure that certain word appears in both code word corpora as well as plain text corpora. Having such sentences will test the true extent to which LLMs can distinguish between sentences containing code words and sentences not containing code words.

I then use BigQuery to run an SQL query to fetch comments containing these code words. An illustration of how a simple SQL query on BigQuery looks like is shown below:

A screenshot of a computer

Description automatically generated

Figure ‑ BigQuery Workspace to run SQL queries

The SQL query in the figure Figure 3‑1 BigQuery Workspace to run SQL queries tries to look for the word “coke” in the comments of subreddit “DarkNetMarkets” posted during the period of August 2016. A longer timeframe will have to be considered to obtain sentences containing all the code words in our list. This process has been automated using the APIs provided by Google BigQuery platform. Using these APIs, I can run a loop and execute the query for each code word, fetch the sentences containing the code words and log the results in a file which can be used for testing later. Below is code block from the program which has been created to invoke the BigQuery API and generate the code word corpora:

A screen shot of a computer program

Description automatically generated

Figure ‑ - Code Screenshot- BigQuery API call

The program parses through the file containing names of drugs and their known code words. For each of these code words, the program then prepares an SQL query and then invokes the BigQuery API. The query results returned by the API are then logged in a file. Multiple iterations of this script were executed by modifying the timeframe of the query and multiple files were obtained as a result of these iterations.

Another program was then created to merge the contents of all these files into a single consolidated file which contains sentences grouped together based on the code words they contain. Below is a screenshot of the code block which does the merging of these files.

A screen shot of a computer program

Description automatically generated

Figure ‑ Code screenshot - Merge code word files

### Preparation of the plain text corpora

To prepare the plain text corpora a few other normal words were chosen to be queried from a popular subreddit called “LifeProTips”. The words chosen were once again carefully picked to make sure that such words also appear alongside code words in sentences containing code words.

The list of words chosen is as follows:

|  |  |
| --- | --- |
|  | Ice,Ice cream, Flour,Cactus, Elephant, Salt, Amoeba, Eat, drink, crime, danger, smoking, inject, smoke, take, jump, sleep, fight, travel,pills, dream, medicine, pharma |

Table ‑ Normal Word list for preparing plain text corpus

A program similar to the one used to generate code word corpus was used for generating the plain text corpus as well, a screenshot of the code block from the program is shown below:

A screen shot of a computer program

Description automatically generated

Figure ‑ Code screenshot- Prepare plain text corpus

This program was once again run for multiple iterations with different time frames and all the files generated from each iteration were then merged using a program similar to the one used to merge the code word corpus files shown in section 3.2.1.

### Dataset consolidation

After having prepared the code word corpora and the plain text corpora using the steps described above, the next step is to consolidate both of them into a single test dataset. Manual intervention was required in this step, to pick sentences that meet the necessary criteria and to label each of the sentences as containing code words and not containing code words. A total of 206 sentences were picked from both plain text corpora and code word corpora and they were all labelled appropriately. This dataset is balanced with an equal number of sentences containing code words and sentences not containing code words.

### Data Cleaning

The test data prepared above needs to be validated for the presence of appropriate labels and it has to be shuffled before it can be used for testing.

The first step is to verify if all the sentences contain a label indicating the presence or absence of code words. This was done using the code block below:

A computer screen with text on it

Description automatically generated

Figure ‑ Data cleaning: Check for the label "Yes" or "No"

Sentences that were missing appropriate labels for the presence or absence of code words were corrected at this step.

The next step is to verify if the sentence labeled to be containing a specific code word, contains that code word in it. That is if a sentence is labeled to be containing the code word “coke”, we need to verify if the sentence contains this word.

This was done using the code block below:

A computer screen with text on it

Description automatically generated

Figure ‑ Data cleaning: Check if the sentence contains a code word identified by the label

Sentences with improper code word labels were corrected at this step.

The final step is to prepare a data frame containing the sentences and their corresponding labels for easier data processing during testing. This data frame was also shuffled and then saved to a file. The code blocks for doing the same are shown below:

A screen shot of a computer program

Description automatically generated

Figure ‑ Data Cleaning: Preparing data frame

### Exploratory Data Analysis

The final dataset used for testing the model has four columns, the “Sentences” column contains sentences containing code words, the “Present” column indicates whether the sentence contains code word or not, the “Code Word” column contains the code word if a sentence contains a code word, else it has a Null value for normal sentences, the “Target Word” column contains the target word corresponding to the code word in the previous column. The first 5 rows of the dataset is shown below, here the dataset has been loaded as a pandas data frame.

A screenshot of a computer

Description automatically generated

Figure ‑ EDA: First five rows.

The dataset has 206 rows, and it has 103 sentences containing and 103 normal sentences. Only half of this dataset has been used to test the Large Language Models. This resulting dataset may not be balanced. But since we are not training or fine-tuning any of the Large Language Models, an imbalanced dataset will not adversely affect the testing, as we only use this dataset for testing the model through prompts. The dataset was prepared by me and it was already cleaned in the previous section, so there is no need for further cleaning to remove duplicates or handling null values.

## Experiments with GPT

OpenAI provides APIs for interacting with their state-of-the-art GPT models described in section 2.8 . These APIs simplify the process of employing GPT-3 for custom tasks such as code word detection. There are other APIs that are meant for fine-tuning the pre-trained models on custom task-specific data, as well as APIs that enable the generation of word embeddings for a given list of words (OpenAI, 2023).

There are two major endpoints that can be used for text generation and question-answering types of tasks, they are the “completion” endpoint and “chat-completion” endpoint.

The “completion” endpoint is a legacy API that takes a single string as an input and generates output. The most advanced model available in this endpoint is “text-davinci-003” (OpenAI, 2023)

On the other hand, the “chat-completion” endpoint is more suited for few-shot prompting due to the capability to assign roles to the prompts and the provision for setting an initial context for the task. The most advanced model available in this endpoint is the “gpt-3.5-turbo”, which is an upgrade on previous GPT models such as “text-davinci-003” (OpenAI, 2023)

For this research, after experimenting with both endpoints, I have opted to use chat-completion API along with the “gpt-3.5-turbo” model because of the better results achieved during initial testing.

### Determination of the optimum prompts for GPT-3

Several experiments were conducted with different types of prompts to identify the prompt that is best suited for code word detection tasks. A prompt message usually consists of an array of individual messages, each having its own specific role. There are 3 distinct roles that can be used- system, user, and assistant. The system role is used to specify a context and influence the behaviour of the model. The user role is used to indicate that the content of the message corresponding to this role was input by the user. The assistant role is used to indicate that the message with this role is the response obtained from the GPT model (*GPT - OpenAI API*, no date)

In the case of the zero-shot prompting technique, there would be a message object with a system role, followed by a message object with a user role. This whole prompt message is then sent to GPT by invoking the API and the response message received will have an assistant role (*GPT - OpenAI API*, no date). An example of zero-shot prompt for code word detection is shown in the screenshot below:

A screenshot of a computer

Description automatically generated

Figure ‑ Zero-Shot prompt example

In Figure 3‑9 Zero-Shot prompt example shown above, the message with the assistant role is the response from the GPT model. Using a message with a system role we set the context of our task, and the message with task-related prompt has a user role. We can observe from the response that the prompt needs to be improved in order to get responses that are consistent and follow a fixed pattern.

In the case of a one-shot prompting technique, there would be a message object with a system role, followed by a message object with a user role, and this is then followed by a message object with an assistant role. These first two messages, one with a user role and another with an assistant role serve as an example to the GPT model. The actual prompt with the custom task is then appended to this message array with a user role. The whole prompt message is then sent to GPT by invoking the API and the response message received will have an assistant role (*GPT - OpenAI API*, no date). An example of one shot prompt for the code word detection task is shown below:

A screenshot of a computer

Description automatically generated

Figure 3‑10 One-shot prompt example

In the Figure 3‑10 One-shot prompt example shown above, the text highlighted in yellow, is the response from the GPT model and it has an assistant role. The first message with a user role followed by a message with an assistant role is the example prompt and the subsequent user message is the actual task-related prompt. We can observe from the response that although it is in the desired format, the model does not know what format to use in case there is no code word in the sentence. This problem can be solved using the few-shot prompting technique.

In the case of the few-shot prompting technique, the message follows the one-shot prompt described above, but instead of just a single example, there will be two or more example prompts preceding the actual task-related prompt. This implies that for each example prompt, there exists one message with a user role and another message with an assistant role. An example of the few-shot prompt for code word detection task is shown below:

A screenshot of a computer

Description automatically generated

Figure 3‑11 Few-shot prompt example

In the Figure 3‑11 Few-shot prompt example shown above, we can observe that it is possible to get GPT to identify code words in multiple sentences at the same time using appropriate prompts with the few-shot prompting technique.

However, there are several challenges when working with GPT models. All of the above screenshots are from the Playground section of OpenAI’s website and when we use similar prompts with their API using OpenAI’s Python library, the results are often not consistent with what we observe on the website. This is due to various factors such as the temperature setting, presence or absence of additional space, and newline character in the prompts. Another challenge of using prompts with multiple sentences is that the order of sentences and the length of sentences will alter the results. A screenshot of this problem is shown below where for a set of 5 sentences, the response from GPT changes based on the order of sentences and length of sentence. In these screenshots, we can notice that when re-executing the test with the same prompt, the response from GPT is different. Also, note that this screenshot is from the Python implementation of invoking GPT API, the code blocks will be explained later on in the upcoming sections.

A screenshot of a computer program

Description automatically generated

Figure 3‑12 GPT responds with "No" for sentence 3, 4

A screenshot of a computer

Description automatically generated

Figure 3‑13 GPT responds with “No” only for sentence 3 on re-execution

Due to the aforementioned reasons for this research, I have conducted the experiment with only one sentence in the task-related prompt. After multiple iterations of testing, the most optimum prompt was identified as below.

**System message:** {"role": "system", "content": "You are an investigator looking for code word for drugs hidden in sentences. Learn from the given examples and answer with a No if code word is not present. If code word is present, identify code word and what it refers to"}

**Two example prompts:**

**User message:** {"role": "user", "content": "Lol, that shit is probably so stepped on you can't even call it coke anymore."},

**Assistant message**: {"role": "assistant", "content": "Present: Yes, Code word : coke, Code word meaning : Cocaine"},

**User message:** {"role": "user", "content": "No one would resist a pot of soup"},

**Assistant message**: {"role": "assistant", "content": "Present: No,"}

The task-related prompt is then appended to this array of messages with a user role and then sent to GPT using the API.

### Testing GPT-3 using OpenAI API and few-shot prompting technique

Using the dataset created in the 3.2.4 section, and the prompts finalized in the previous section, I have tested the code word detection capabilities of GPT-3 using their chat-completion API endpoint.

To begin with, I must set the OpenAI API key which I obtained from the OpenAI website during the registration of my account. This step is necessary in order to be able to invoke the API. Followed by this I have the model to be “gpt-3.5-turbo” and the template of a fixed few-shot prompt is also created and stored as an array of messages. Individual sentences from the test dataset will then be appended to this array before API invocation. A screenshot of the same is shown below:

A screenshot of a computer program

Description automatically generated

Figure 3‑14 GPT-3 testing: set API key, prompt template

Followed by this I have created a few helper functions to log some information in text files as well as a function to invoke the OpenAI API. The function passes the chosen model name, the final prompt message as well the temperature setting during the API invocation. The temperature setting determines the degree of randomness of the response, a lower value signifies a more focused, deterministic, and consistent response. In our experiment, the temperature is set to 0 (OpenAI, 2023). This function also tries to capture all possible exceptions that can occur during API invocation. The function then returns a response message from API in case of a successful response. In case of an exception, an appropriate error message will be returned. The screenshot of this code block is shown below:

A screen shot of a computer program

Description automatically generated

Figure 3‑15 GPT-3 Testing : Function to invoke API

The next section is the main driver code for this testing. Here we iterate through each row of the dataset and pick individual messages and append it to the prompt template. This is then passed to the function to invoke the API. The response from GPT is then logged into an output file. While logging, a pipe (“ | “) delimiter is used, so that the file can be loaded and operated as a data frame, using pandas functions, for evaluating the result of testing. A screenshot of the driver function is shown below:

A screen shot of a computer program

Description automatically generated

Figure 3‑16 GPT-3 testing: Driver code

## Experiments with PaLM 2

PaLM2 described in section 2.9 can be accessed using Vertex AI, which is a machine learning platform that simplifies the process of training large models and development of AI applications. Vertex AI is part of a vast number of other applications available on the Google Cloud platform. Vertex AI provides a user interface to interact with several Large Language Models through its Generative AI Studio section. These models can also be accessed using Vertex AI APIs. (Introduction to Vertex AI  |  Google Cloud)

There are two different variations of PaLM2 based on the type of task, they are – PaLM 2 for Text and PaLM 2 for Chat. PaLM 2 for chat is suitable for tasks that require a conversation type of interaction with the model. For other types of tasks, PaLM 2 for Text is ideal. (*PaLM 2 for Text – Vertex AI – ResearchProposal – Google Cloud console*, no date)

For this research, I have used the Vertex AI API for PaLM 2 for Text with the “text-bison” model.

### Determination of the optimum prompts for PaLM 2 for Text

Similar to GPT-3 model, PaLM 2 also works best with the few-shot prompting technique. Before arriving at this conclusion, I conducted experiments with zero-shot prompts and one-shot prompts using prompt examples used for GPT-3 in section 3.3.1. The primary difference between GPT-3 prompts and prompts for PaLM2 is the simplified prompt format for PaLM2, in which there is no concept of assigning a role to each sentence. Instead, the prompts just begin with “Q” to indicate a “Question” and “A” to indicate an answer. In the case of example prompts, each example consists of a “Question” prompt followed by an example “Answer” prompt. Apart from Question and Answer prompts, there is also an initial “Context” message which is used to set the context and control the model’s behaviour and response. This is similar to the “System” role message used in prompts for GPT-3 prompts. An example of a few-shot prompt is illustrated in the screenshot below. The screenshot is taken from Vertex AI’s Generative AI Studio.

A screenshot of a computer

Description automatically generated

Figure ‑ PaLM 2 Few-shot prompt example

The final prompts used for code word detection task is as follows:

*You are an investigator looking for code words for drugs hidden in sentences. You have to analyze the sentences and determine whether they contain a code word for drugs. If the sentence contains a code word for drugs, answer as Present: yes and identify the code word in the sentence. If the sentence does not contain any code word for drugs then answer as Present: no.*

***Q****: Lol, that shit is probably so stepped on you can\'t even call it coke anymore.*

***A****: Present: Yes, Code word: coke, Code word meaning : Cocaine*

***Q****: No one would resist a pot of soup*

***A****: Present:No*

***Q****: My cousin did the same and when they legalized pot in dc they really started cracking down in virginia and maryland.*

***A****: Present: Yes, Code word : pot , Code word meaning : Marijuana*

***Q****: i understand this is to get more customers but imo its bullshit*

***A****: Present:No*

The first paragraph is the context prompt, which is followed by few-shot examples which contain a Question-and-Answer pair in each example. This is then followed by the actual task-related prompt beginning with a “Q”. The model then responds back with an appropriate answer that begins with “A”.

### Testing PaLM 2 with Vertex AI API and few-shot prompting technique

Using the same dataset which was used for GPT-3 testing, I have tested the code word detection capabilities of PaLM 2 using Vertex AI’s API. To use VertexAI APIs we need to set up Vertex AI SDK for Python. For this experiment, I have used Google Collab due to the simplicity of setting up Vertex AI SDK on Google Collab and also because of the cross-platform authentication possible when using the same Google account for both Vertex AI and Google Collab.

To begin with, I have installed the prerequisite libraries required to use Vertex AI API on Google Collab. Next, I have defined a few API parameters such as temperature, project name, max\_ouput\_tokens, and finally the name of the model which is being used in the API. The model used is “text-bison@001”. A screenshot of this code block is shown below:

A screenshot of a computer program

Description automatically generated

Figure 3‑18 PaLM2 API parameters

After this I have defined the prompt template with few-shot examples. Each individual sentence in the dataset, will then be appended to this few-shot prompt. Next, I have created a few helper functions to format the sentences, log the text generated into a file, and finally a function that calls the Vertex AI API and returns the response from API. A screenshot of this function is shown below:

A computer screen with text and symbols

Description automatically generated

Figure 3‑19 PaLM2: Function to call Vertex AI API

Finally, we have the main driver code which is similar to the driver code used for testing GPT-3 which was explained in detail earlier. However, there are a few minor changes. The number of API requests that can be made to the “text-bison” model is restricted to 60 calls per minute (*Quotas and limits  |  Vertex AI  |  Google Cloud*, no date). Due to this restriction, after multiple iterations of testing, I found that waiting for 5 seconds before making each API call resulted in the successful completion of testing all sentences without any API quota exceptions. A screenshot of the main driver code is shown below:

A screen shot of a computer program

Description automatically generated

Figure 3‑20 PaLM 2 testing : Driver code

This driver code generated an output file, which contains all the sentences used for testing along with their corresponding response from PaLM 2 model.

## Experiments with Zero-shot classification and small LMs

In this section, I will describe the experiments carried out using two different zero-shot classification models described in section 2.10. They are:

* facebook/bart-large-mnli
* MoritzLaurer/DeBERTa-v3-base-mnli-fever-anli

The generative language models like GPT-3 and PaLM 2 described above are instruction tuned and have in-context learning capabilities to perform a wide range of tasks such as feature extraction, reasoning, and classification among many other tasks. Due to these reasons, such models were able to detect the presence of code words, identify the code words and finally also predict what actual word, each code word refers to. On the other hand, we have zero-shot classification models that can classify sentences into different candidate labels that are passed to them as input. The basic premise for these models is explained in section 2.10

For example, consider that we provide a sentence and a set of candidate labels to the zero-shot classification model as below:

Sentence: *I ordered a device from your website, but that doesn’t seem to be working, I would like to be reimbursed.*

Candidate Labels: *[“refund”, “faq”, “review”, “reorder”]*

The model will then give a response containing an array of values where each element of the array represents the predicted probability of the sentence belonging to this label. For the example given above, the response from facebook/bart-large-mnli is as follows:

A white background with a purple line

Description automatically generated

Figure ‑ Zero-shot classifier example

From the screenshot, we can observe that the label “refund” has the highest probability, implying that according to the model’s prediction, the sentence belongs to the class “refund”.

I have used the same principle to classify sentences into two classes, i.e., “sentences containing code words for drugs” and “normal sentences”.

To interact with these models, I have used the Hugging face inference API (Hosted Inference API). These APIs from hugging face are free to use and provide access to thousands of open-source models for testing and evaluation. These APIs save us from the hassle of having to download the parameters of the transformer models to our local machine to test and evaluate large language models.

### Determination of the optimum prompts for zero-shot classification LMs

As the name of the models suggests, these models are meant to be used without providing any examples. The format of the prompt is straightforward and consists of an input element to which the sentences from the dataset are passed one after another. Followed by the input element, we have the parameters element in which we have to pass the candidate labels as one of the parameters. The actual prompt used is as shown below:

*{*

*"inputs": "Lol, that shit is probably so stepped on you can't even call it coke anymore.",*

*"parameters": {"candidate\_labels":* [*"sentence with code word for drugs", "normal sentence"]}*

*}*

As we can see from the prompt above, I am only passing two candidate labels to the model for classification.

### Testing zero-shot Classification small LMs

Using the same dataset that was previously used for testing GPT-3 and PaLM 2, I have tested two different zero-shot classification models. They are facebook/bart-large-mnli and MoritzLaurer/DeBERTa-v3-base-mnli-fever-anli. Since they are both zero-shot classification models, the input prompt format as well as the output response format for both models remain the same. Hence, I have used the same code with a few minor changes for testing both these models.

To begin with I have to set the API token which will be used for authorizing the requests from my account. The API token was obtained from the Hugging Face website and is linked to a specific user account. Followed by which I set the API URL that has a model card that is unique to each model on the Hugging Face website.

Next, I have created a few helper functions for logging text data to files as well as a function to invoke the hugging face API. This function also captures any API exceptions that can occur and returns either a JSON response or an Error message depending on whether the API call was successful or failed. The code block for this function is shown below:

A computer screen shot of a program code

Description automatically generated

Figure 3‑22 Zero-Shot classifier testing: Function to call Hugging Face API

After this is the parameters for the API are defined as this remains constant for every API call. The main parameter is the “candidate label”, to which I assign two values - "sentence with a code word for drugs" and "normal sentence". This parameter is appended to the JSON body along with a single sentence from the dataset in a prompt format described in the previous section. The output of the model consists of two arrays, one for scores and another for labels. The driver code then parses the response to check the index of the array having the highest probability in the scores array. Using that index we then fetch the corresponding element in the labels array from the response. If the label “sentence with a code word for drugs” has the highest probability in the response, then we consider that the sentence was predicted to be belonging to this class, otherwise, we consider that the sentence was predicted to be a normal sentence. These observations are then logged into a file which will be used for evaluation later. There is also a line of code added to wait for 10 milliseconds before making the next API call to prevent overloading of the server as per the API calling guidelines on the Hugging Face website (Hosted Inference API). A screenshot of the main driver code is shown below:

A screen shot of a computer program

Description automatically generated

Figure 3‑23 : Zero-shot classifier testing: Driver code

## Experiments with FastText classifier

For this experiment, we have to train a model using the fastText library described in section 2.4, and run predictions using this model on our dataset. The training and testing were carried out on Google Collab due to the ease of setting up the prerequisite libraries required to install the FastText library (Joulin *et al.*, 2016).

The training of the model was done using the dataset which was prepared for this project initially. Only half of this dataset was used for testing GPT-3, PaLM 2, and zero-shot classification models. The other half of this dataset which was unused so far was used to train the FastText model. The testing of this model was done using the same dataset which was used for testing all other models previously discussed.

### Dataset Preprocessing

Before training the model, all the sentences in the dataset have to be pre-processed. Although tokenization is not necessary as per the FastText documentation, we still have to preprocess the dataset to remove, special characters, upper case letters, and numerical values (Joulin *et al.*, 2016). I created a function for this, and this function was applied to the whole dataset. A screenshot of this function is shown below:

A screenshot of a computer

Description automatically generated

Figure ‑ FastText Preprocessing Dataset

### Dataset Preparation

To train the model the sentences in the dataset should follow a specific format, i.e. each line should contain a sentence along with a set of candidate labels. Each label should have the “\_\_label\_\_” word prepended to it. For my experiment, each line of the dataset looks like the following:

*“\_\_label\_\_yes \_\_label\_\_grass smoking grass while cutting grass* “

In this above example, there are two labels, one for the presence or absence of a code word and another label indicating the type of code word present in the sentence.

After creating this dataset, the results are then stored in two separate files, one file will be used for training and another file will be used for testing. This is because the FastText library only accepts a file as input for training.

### Training of the FastText Model

Once the preparation of the dataset is complete, I installed the necessary FastText libraries and trained the model by passing the file with training data as input. The code block for this is shown below:

A black background with colorful lights

Description automatically generated

Figure 3‑25 FastText Model training

For training, I have set a few hyperparameters as well. The optimal value for these hyperparameters was determined after multiple iterations of testing with different possible values. The final values used for training are as follows, the learning rate(lr) is set to 0.7, the number of epochs(epoch) for training is set to 25, the maximum length of word ngram (wordNgrams) is set to 3, the size of the word vectors(dim) is set to 50, the number of buckets used is set to 200000 and the loss function(loss) used is ‘ova’ which stands for One vs All reduction strategy and is usually used for multi-class classification problems (Brownlee, 2021)

The Learning rate is a hyperparameter that determines the step size at each iteration of training while trying to reduce the value of loss function (Zulkifli, 2018)

The number of epochs controls the total number of times the training dataset is passed through the entire neural network during the process of model training (SHARMA, 2017)

The maximum length of word n-gram determines the maximum value for n-grams which are explained in detail in section 2.4

The dimension(dim) hyperparameter determines the dimension of the word embedding vector generated for each word using the n-grams.

A loss function is a function that measures the discrepancy between model’s prediction and actual truth value. There are several algorithms to compute this loss function, the one which I have used above is Ova. One vs All loss function is used in multi-class classification problems and involves training multiple binary classifiers in which each of this binary classify is able to differentiate between a single class from rest of the classes (Brownlee, 2021)

### Testing of the FastText Model

Once the training of the FastText model is complete we can evaluate the results using two methods that are available in the FastText library.

The “predict” methods can be used to predict the classification of individual sentences or a list of sentences and the model will output a list of predicted labels and their corresponding predicted probabilities (Joulin *et al.*, 2016). A simple illustration of this is shown in the screenshot below:

A black screen with a black background

Description automatically generated with medium confidence

Figure 3‑26 FastText predict method example

We can observe from the output that the label “yes” has the highest probability. This implies that the model classified the sentence as a sentence containing code words for drugs.

Another method available is the “test” method which takes a file containing a test dataset as input and evaluates the whole test dataset and outputs precision and recall (Joulin *et al.*, 2016). The dataset which was used to evaluate all the other previously described models was used to evaluate the FastText model as well. However, the test dataset is also pre-processed and modified as explained in the Dataset preprocessing and Dataset preparation sections of this chapter. Below is a screenshot of the code block that performs testing and the corresponding output from the model which shows the number of samples, precision, and recall.

A screen shot of a computer

Description automatically generated

Figure 3‑27: FastText: precision and recall

Further discussion on the evaluation of these results is performed in section 4.5.

# Evaluation of Results

This chapter presents a detailed evaluation of the results of all the experiments conducted in Chapter 3.

## Evaluation Metrics Used

For evaluating the results of all the models, I have used 3 of the most common evaluation metrics used in machine learning, they are, precision, recall and accuracy.

Precision measures, out of the total positive predictions made, how many were actually positive (Shung, 2018). It is given by the formula:

A diagram of positive and negative

Description automatically generated

Figure ‑ Formula for Precision(Shung, 2018)

Recall measures, the ability of the model to correctly identify the positive instances (Shung, 2018). It is given by the formula:

A close-up of words

Description automatically generated

Figure ‑ Formula for Recall(Shung, 2018)

Accuracy is a measure of total number of correct predictions out of the total number of samples tested (Harikrishnan N B, 2019). It is given by the formula:

A black text with a plus and a line

Description automatically generated

Figure ‑ Formula for Accuracy (Harikrishnan N B, 2019)

In the Figure 4‑3 TN stands for True Negative, TP stands for True Positive, FP stands for False Positive and FN stands for False Negative.

I have also used a confusion matrix in my evaluation which is a table that summarizes the model’s predicted values and actual values (Harikrishnan N B, 2019). What each row and column of the confusion matrix signifies is shown below:

A diagram of negative and false positive

Description automatically generated

Figure ‑ Confusion Matrix Columns(Harikrishnan N B, 2019)

## Evaluation of GPT-3 Testing results

To begin with, we will evaluate the accuracy of GPT-3 at predicting whether a sentence contains code words or not. The evaluation of results begins with loading the original file containing the truth values as well as the output generated by GPT as shown below:

A black screen with white text

Description automatically generated

Figure ‑ GPT Evaluation: Load files

Next, I have extracted 50% of rows from the original test dataset because GPT-3 was also tested only for 50% of sentences in the dataset. After that, I converted the values “yes” and “no” to integer values 1 and 0 respectively. Finally, I have used scikit learn libraries to generate a confusion matrix, classification report, and accuracy score. A screenshot for these code blocks is shown below:

A screenshot of a computer program

Description automatically generated

Figure 4‑6 GPT-3 evaluation: code for confusion matric and classification report

The confusion matrix for the results of the testing has been depicted using a heatmap shown below:

A screenshot of a computer screen

Description automatically generated

Figure ‑ GPT-3 evaluation: confusion matrix

The columns of the confusion matrix have been explained in section 4.1 using Figure 4‑4. In the confusion matrix, we notice that there are 8 False Positives and 2 False Negative cases. A closer analysis of these False positives will reveal the scenarios where GPT-3 wrongly predicted a normal sentence as a sentence containing code words. The 8 false positive cases are shown below:

A screenshot of a computer

Description automatically generated

Figure ‑ GPT Evaluation: False Positives

We can notice from the above figure that when the sentences contain word such as “drugs”, “pills”, “smoking”, GPT-3 wrongly considers them to be code words. This could be because such words usually appear alongside sentences containing drug related words and code words. In other cases, words like “gold dust” and “coca-cola” are actual drug code words, however, in the above sentences they appear as normal words, but GPT-3 mistakes them to be code word for drugs. In case of “coca-cola”, the GPT-3 seems to be referring to the beverage, based on the predicted target code word. This is an anomaly because, the model has clearly mistaken the drug code word “coca-cola” with the beverage “coca-cola”. Another explanation to this could be that as an autoregressive language model, GPT-3 only generates the next word in a sequence based on the probability distribution for each word and the model included the word “beverage” in the predicted target word column due to its autoregressive nature.

Below are the results of classification report and accuracy report for the same task:

A screenshot of a computer screen

Description automatically generated

Figure 4‑9 : GPT-3 evaluation: and classification report

We can observe that GPT-3 has a high overall accuracy of 90% when it comes to identifying whether sentences contain code words for drugs or not.

Next, we need to evaluate whether the code words identified by GPT are accurate or not. Below is a table that shows the total number of correct predictions for each code word:

|  |  |  |
| --- | --- | --- |
| Code word | Total Samples | Correct Predictions |
| crystal | 3 | 3 |
| baby powder | 5 | 0 |
| fish scale | 3 | 3 |
| acid | 3 | 3 |
| haze | 2 | 1 |
| coke | 3 | 3 |
| chiva | 3 | 3 |
| candy | 2 | 0 |
| blue jeans | 2 | 2 |
| snowflake | 1 | 1 |
| ganja | 2 | 1 |
| weed | 1 | 1 |
| love drug | 1 | 1 |
| la tierra | 2 | 2 |
| kleenex | 1 | 0 |
| dust | 2 | 2 |
| china white | 2 | 2 |
| gold dust | 1 | 1 |
| pot | 1 | 1 |
| brown sugar | 1 | 1 |
| vitamin k | 2 | 1 |
| grass | 1 | 1 |
| barry farrell | 1 | 1 |

Table ‑ GPT-3 Code word Predictions

Overall accuracy was calculated using the formula overall correct predictions/total samples. The overall accuracy was found to be 75.66 percent. This was calculated using a simple function that builds a dictionary of total samples and total correct predictions for each code word. Finally, the function sums up these two values to find the overall accuracy. We can notice although GPT-3 has an accuracy of 90% when it comes to predicting whether a sentence contains a code word or not, the accuracy drastically decreases with respect to the code words identified by GPT-3 in those sentences. Words like “baby powder”, “candy” and “Kleenex” have not been identified as code words by GPT-3.

Finally, we will look into the target words identified by GPT-3. In the case of the target word, we cannot do an exact word-to-word comparison because GPT-3 uses synonyms or sentences to convey what the code word implies. For example, GPT-3 identifies “amnesia haze” to be a code word for “a specific strain of cannabis”. Although this can be restricted based on modifying the prompt there is another example where “crystal” was identified to be a code word for “Crystal methamphetamine”. Due to this reason, I have manually analysed the results and found that GPT-3 has correctly identified the target word in most cases when it has already correctly identified the code word.

## Evaluation of PaLM 2 Testing results

Since the output file generated by PaLM 2 has the same format as that of the output file generated after GPT-3 testing, evaluation of the results was done using the same technique which was used to evaluate GPT-3 test results. To begin with, I have evaluated the accuracy of PaLM 2 at predicting whether a sentence contains code words or not. The confusion matrix for the results of the testing PaLM 2 has been depicted using a heatmap shown below:

A screenshot of a graph

Description automatically generated

Figure ‑ PaLM 2 evaluation: Confusion Matrix

In the confusion matrix we can observe that there are 4 False Positive and 8 False Negative cases. The False Positive cases are almost similar to the GPT-3’s results where sentences containing words “smoking”, “drugs”, “ambien” and “epoxy” were wrongly predicted to be sentences containing code words. However, the False Negative cases are more concerning since the model failed to detect code words in those sentences. The False Negative cases are shown below:

A screenshot of a computer

Description automatically generated

Figure ‑ PaLM 2 evaluation: False Negative Cases

GPT-3 failed to identify the word “baby powder” as a code word too. However, PaLM 2 has failed to identify code words in other sentences containing words such as “fish scale”, “blue jeans”, “brown sugar”.

Below is the screenshot which shows the precision, recall, and accuracy of PaLM 2 at this task:

A screenshot of a computer screen

Description automatically generated

Figure 4‑12 PaLM2 evaluation : Confusion matrix, classification report

We can observe that PaLM 2 has an accuracy of 88% when it comes to predicting whether a sentence contains a drug-related code word or not.

Next, we evaluate the accuracy of the code words identified by PaLM 2. For this, I have again used the same technique used for GPT-3 evaluation Overall accuracy was found to be 64% for the task of identifying code words for drugs in sentences. We can notice that the accuracy has drastically decreased when it comes to identifying the code words in sentences predicted to be containing code words. PaLM 2 has failed to recognize words such as “baby powder”, “candy”, “brown sugar”, “Kleenex” and “vitamin k” as code words. Although GPT-3 model also missed some of these words, PaLM 2 model failed to recognize more words when compared to GPT-3 model.

Below is a table that shows the total number of correct predictions for each code word:

|  |  |  |
| --- | --- | --- |
| Code word | Total Samples | Correct Predictions |
| crystal | 3 | 3 |
| baby powder | 5 | 0 |
| fish scale | 3 | 2 |
| acid | 3 | 3 |
| haze | 2 | 1 |
| coke | 3 | 3 |
| chiva | 3 | 3 |
| candy | 2 | 0 |
| blue jeans | 2 | 1 |
| snowflake | 1 | 1 |
| ganja | 2 | 2 |
| weed | 1 | 1 |
| love drug | 1 | 1 |
| la tierra | 2 | 1 |
| kleenex | 1 | 0 |
| dust | 2 | 1 |
| china white | 2 | 2 |
| gold dust | 1 | 1 |
| pot | 1 | 1 |
| brown sugar | 1 | 0 |
| vitamin k | 2 | 0 |
| grass | 1 | 1 |
| barry farrell | 1 | 1 |

Table 4‑2 Evaluation of PaLM2: Code word detection accuracy

Finally, we will look into the target words identified by PaLM 2. Due to the same reasons previously explained for GPT-3 target word evaluation, I have manually analysed the results and found that PaLM 2 model has correctly identified the target word in most cases when it has already correctly identified the code word.

## Evaluation of zero-shot classification LLMs

Unlike GPT-3 and PaLM 2, zero-shot classification models can only classify whether a sentence contains code words or not. Hence, in this section, we will only check the accuracy of the models in classifying the sentences for the presence or absence of code words. The evaluation technique for this is similar to the one used for evaluating GPT-3 and PaLM 2. We begin by loading the original dataset along with the two output files generated by the two models, one from facebook/bart-large-mnli and another from MoritzLaurer/DeBERTa-v3-base-mnli-fever-anli as shown in the screenshot below:

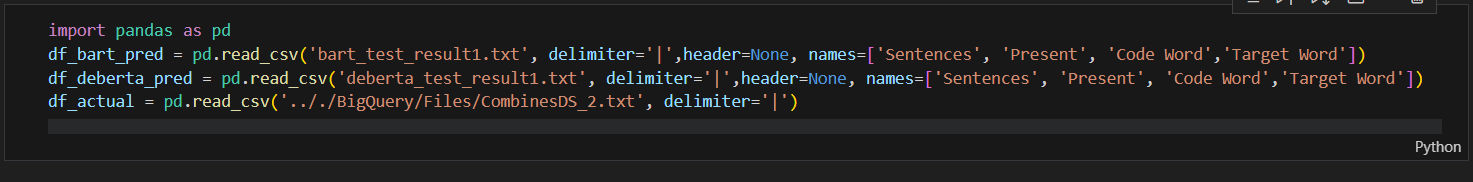


Figure 4‑13 Zero-Shot classifier evaluation: load files

After this, I converted the label “yes” to 1 and “no” to 0 and calculated the confusion matrix, classification report, and accuracy using sci-kit learn libraries.

Below is a screenshot of the confusion matrix for **facebook/bart-large-mnli** model:

A screenshot of a computer screen

Description automatically generated

Figure ‑ facebook/bart-large-mnli: Confusion Matrix

We can notice a high number of False Positive cases in the confusion matrix which indicates that the facebook/bart-large-mnli model misclassified a large number of normal sentences in the test dataset as sentences containing code words.

Below is a screenshot of the classification report:

A screenshot of a computer screen

Description automatically generated

Figure 4‑15: facebook/bart-large-mnli - classification report

Similarly, below is the screenshot of confusion matrix MoritzLaurer/DeBERTa-v3-base-mnli-fever-anli.

A screenshot of a graph

Description automatically generated

Figure ‑ MoritzLaurer/DeBERTa-v3-base-mnli-fever-anli confusion matrix

We can notice once again notice further degradation in the result quality with high number of False Positive cases in the confusion matrix which indicates that the MoritzLaurer/DeBERTa-v3-base-mnli-fever-anli model misclassified a large number of normal sentences in the test dataset as sentences containing code words.

Below is the classification report for MoritzLaurer/DeBERTa-v3-base-mnli-fever-anli:

A screenshot of a computer screen

Description automatically generated

Figure 4‑17 : MoritzLaurer/DeBERTa-v3-base-mnli-fever-anli confusion matrix and classification report

We can observe from the screenshots that facebook/bart-large-mnli has an accuracy of 70% while MoritzLaurer/DeBERTa-v3-base-mnli-fever-anli has an accuracy of 55%. This indeed proves that zero-shot classification models such as facebook/bart-large-mnli can be used for code word detection tasks. They may serve as a first-level filter for moderators to flag sentences that potentially contain code words. This performance can be further improved by fine-tuning these models further on a dataset containing sentences with code words with drugs.

## Evaluation of FastText model results

The values for precision and recall obtained in section 3.6.4 are when the problem is treated as a multi-class classification problem. The accuracy of predicting the code words within sentences is very low and hence I have manually calculated the accuracy by using the “predict” function to predict a class with the highest probability and then used these values along with sci-kit learn libraries to calculate the precision and accuracy. Below is a screenshot of the code block which first writes the predicted labels to a file and then converts them to numerical values for 1 for “yes” class and 0 for “no” class.

A screenshot of a computer program

Description automatically generated

Figure 4‑18 FastText Convert predicted labels to numerical values

After this, the truth values were retrieved from the original dataset and then using the actual values and predicted, the confusion matrix, classification report and accuracy was calculated. Below is a screenshot of these metrics, when considering the problem as a binary classification one.

A screenshot of a computer

Description automatically generated

Figure 4‑19 FastText Precision, Recall and Accuracy

From the screenshot we can observe that FastText has an accuracy of 58% when classifying sentences as sentences containing code words for drugs and sentences not containing code words for drugs. From the confusion matrix we can notice that there are a high number of False Positive cases. The accuracy of 58% is marginally better than a random classifier. Such a model requires a larger dataset for training because a lot of the word present in training data may not be present in test dataset. That explains the poor performance of this baseline model.

## Analysis of the performance of all Models

Below table that shows the performance of all models on the task of predicting whether a sentence contains code word or not.

|  |  |  |  |
| --- | --- | --- | --- |
| Model Name | Precision | Recall | Accuracy |
| GPT-3 | 0.84 | 0.96 | 0.9 |
| PaLM 2 | 0.9 | 0.82 | 0.88 |
| facebook/bart-large-mnli | 0.59 | 0.98 | 0.7 |
| MoritzLaurer/DeBERTa-v3-base-mnli-fever-anli | 0.49 | 1.0 | 0.55 |
| FastText | 0.51 | 0.87 | 0.58 |

Table ‑ Performance of All models- Predicting presence of code word

We can observe that GPT-3 outperforms all other models at this task with an overall accuracy of 90%. Although PaLM 2 has higher number of parameters than GPT-3, its performance is slightly lesser than that of GPT-3. But overall, this shows the effect of the scale of LLMs on such tasks. The top two performing models, GPT-3 and PaLM 2 have 175 billion and 540 billion parameters respectively as explained in section 2.7. The LLM which performs the worst is the MoritzLaurer/DeBERTa-v3-base-mnli-fever-anli model and this also happens to have the least number of parameters among all the large language models tested with 184 million parameters (MoritzLaurer/DeBERTa-v3-base-mnli-fever-anli · Hugging Face)

The poor performance of the baseline FastText classifier model can be attributed to the small dataset used for training. Through this I would like to demonstrate that, unlike these models, Large Language Models do not require large task-specific dataset for conducting experiments on Natural Language Processing tasks such as code word detection.

Below is a table that shows the performance of models on the task of identifying the code words present in the sentence.

|  |  |
| --- | --- |
| Model Name | Accuracy |
| GPT-3 | 75.66 |
| PaLM 2 | 64 |

Table ‑ Performance of Models for task of identifying code words

Out of all the models tested, only two models had the capability to identify the code words present in the sentence, while the rest could only predict whether a sentence contains a code word or not. Out of these two models, we can see GPT-3 greatly outperforms PaLM 2 at this task.

Overall, I can conclude that GPT-3 based models are the best suited Large Language Models for code word detection and identification. Although PaLM 2 model’s accuracy is only marginally lesser than GPT-3 with respect to accuracy in task 1 (predict presence of code word in sentence), GPT-3 model’s recall is better than that of PaLM- 2. It is better to False Positives than False Negatives in code word detection tasks because, False Positives can be further scrutinized, but False Negatives mean that we have failed to notice malicious communication altogether.

# Conclusion and Recommendations

As explained in the literature survey of chapter 2, the code word detection capabilities of Large Language Models such as GPT-3 and PaLM 2 were underexplored before my research on this subject. Through my experiments I have proved that autoregressive large language models can be used for code word detection tasks and their accuracy is comparable with the current state-of-the-art. These results were possible without the need for extensive training of the model on a task-specific dataset. The baseline FastText based classifier had poor performance due to the small size of the training dataset. Such models require large text corpus for training and this is a problem that Large Language Models solve. Creating a large dataset that includes enough sentences with code words and normal words is possible, however, the bigger challenge with this is that of the labelling of this big dataset in a format that is accepted by the FastText classifier.

My experiments have also proved that few-shot prompting technique along with the optimal prompts were good enough to instruct models to perform the code word detection tasks and produce output in the desired format.

The smaller language models which are fine-tuned for zero-shot classification such as facebook/bart-large-mnli showed an accuracy of 70% and such models can be fine-tuned further to be used as a first level filter to flag suspicious content, which can later be scrutinized further through human intervention.

The framework used in my experiments can be used to monitor malicious communications in real-time, for this, the only change required is to use a streaming service to feed test data to the model, instead of a text file, which I have used in my experiments.

## Learnings and Experiences

Over the course of this research, I have gained insight on how Natural Language Processing has evolved over the years. I have gained hands-on experience of working with different word embedding techniques such as Word2Vec, GloVe and FastText. Through my experiments I have learned how to use the transformer-based models using Hugging Face libraries, both using their pipeline function as well as their Inference APIs. My experiments have also taught me how to use Open AI APIs and Vertex AI APIs to interact effectively with GPT-3 and PaLM Models. Another valuable knowledge I gained through my experiments is with regards to prompt programming, using which I now know how to effectively designed prompts for Large Language Models.

## Recommendations for Future Work

In my experiments I have used the base version of all the models. However, GPT-3 and PaLM 2 can be fine-tuned on small dataset using their API endpoints meant for fine-tuning them (API Reference - OpenAI API) (Tune language foundation models  |  Vertex AI  |  Google Cloud).

Such fine-tuning will not only improve the performance of these models but will also give greater control over their output format (API Reference - OpenAI API; Tune language foundation models  |  Vertex AI  |  Google Cloud). However, these are not open-source models and certain cost maybe incurred with fine-tuning these models.

But there are other transformer based Large Language Models that are available on Hugging face, which are free to use for non-commercial purposes such as research. Some of these models such as Vicuna (The Vicuna Team, 2023), Falcon (Leandro von Werra *et al.*, 2023) and MPT (*MPT-30B: Raising the bar for open-source foundation models*, no date) have indicated that they are capable of detecting code words to an extent, when I tested them using a web-interface available at websites such as (h2oGPT; Chat with Open Large Language Models). However, conducting a full-scale experiment on such models requires large amount of resources since the model weights easily exceed 15GB and there are no APIs available to interact with them. Given enough resources, experiments can be conducted on these models as well to identify their code word detection capabilities and to take it a step further, such models can be fine-tuned further using a dataset specifically designed to fine-tune these models for code word detection tasks.

# Bibliography

*🤗 Hosted Inference API* (no date). Available at: https://huggingface.co/docs/api-inference/index (Accessed: 10 August 2023).

*500+ Social Media Statistics You Must Know in 2023* (2023). Available at: https://www.socialpilot.co/blog/social-media-statistics (Accessed: 4 August 2023).

Akdogan, A. (2021) *Word Embedding Techniques: Word2Vec and TF-IDF Explained | by Adem Akdogan | Towards Data Science*. Available at: https://towardsdatascience.com/word-embedding-techniques-word2vec-and-tf-idf-explained-c5d02e34d08 (Accessed: 8 August 2023).

Ani Petrosyan (2023) *Internet usage worldwide – statistics & facts | Statista*. Available at: https://www.statista.com/topics/1145/internet-usage-worldwide/#topicOverview (Accessed: 2 April 2023).

*API Reference - OpenAI API* (no date). Available at: https://platform.openai.com/docs/api-reference/fine-tunes (Accessed: 11 August 2023).

Balkus, S. and Yan, D. (2022) ‘Improving Short Text Classification With Augmented Data Using GPT-3’. Available at: http://arxiv.org/abs/2205.10981.

Biderman, S. *et al.* (2023) ‘Pythia: A Suite for Analyzing Large Language Models Across Training and Scaling’. Available at: https://arxiv.org/abs/2304.01373v2 (Accessed: 9 August 2023).

*Bidirectional LSTM in NLP - GeeksforGeeks* (no date). Available at: https://www.geeksforgeeks.org/bidirectional-lstm-in-nlp/ (Accessed: 8 August 2023).

*BigQuery Enterprise Data Warehouse  |  Google Cloud* (no date). Available at: https://cloud.google.com/bigquery (Accessed: 5 July 2023).

Branwen, G. (2020) *GPT-3 Creative Fiction · Gwern.net*. Available at: https://gwern.net/gpt-3 (Accessed: 2 April 2023).

Brown, T.B. *et al.* (2020) ‘Language Models are Few-Shot Learners’. Available at: http://arxiv.org/abs/2005.14165.

Brownlee, J. (2017) *A Gentle Introduction to the Bag-of-Words Model - MachineLearningMastery.com*. Available at: https://machinelearningmastery.com/gentle-introduction-bag-words-model/ (Accessed: 6 August 2023).

Brownlee, J. (2021) *One-vs-Rest and One-vs-One for Multi-Class Classification - MachineLearningMastery.com*. Available at: https://machinelearningmastery.com/one-vs-rest-and-one-vs-one-for-multi-class-classification/ (Accessed: 10 August 2023).

Brownlee, J. (2023) *A Gentle Introduction to Transfer Learning for Deep Learning - MachineLearningMastery.com*. Available at: https://machinelearningmastery.com/transfer-learning-for-deep-learning/ (Accessed: 9 August 2023).

Calderón, F.H. *et al.* (2021) ‘Linguistic patterns for code word resilient hate speech identification’, *Sensors*, 21(23). Available at: https://doi.org/10.3390/s21237859.

CALO Project (no date) *Enron Email Dataset*. Available at: https://www.cs.cmu.edu/~enron/ (Accessed: 5 July 2023).

Cambridge Consultants (2019) *USE OF AI IN ONLINE CONTENT MODERATION 2019, Ofcom Report*.

Catalin Cimpanu (2018) *Reddit Bans Community Dedicated to Dark Web Markets*. Available at: https://www.bleepingcomputer.com/news/security/reddit-bans-community-dedicated-to-dark-web-markets/ (Accessed: 5 July 2023).

*Chat with Open Large Language Models* (no date). Available at: https://chat.lmsys.org/ (Accessed: 11 August 2023).

Chowdhery, A. *et al.* (2022) ‘PaLM: Scaling Language Modeling with Pathways’. Available at: https://arxiv.org/abs/2204.02311v5 (Accessed: 9 August 2023).

Chung, H.W. *et al.* (2022) ‘Scaling Instruction-Finetuned Language Models’. Available at: http://arxiv.org/abs/2210.11416.

*Code word definition and meaning | Collins English Dictionary* (no date). Available at: https://www.collinsdictionary.com/dictionary/english/code-word (Accessed: 4 August 2023).

DEA Intelligence Report (2018) *Slang Terms and Code Words: A Reference for Law Enforcement Personnel DEA Intelligence Brief DEA Intelligence Report*.

Deshmukh, S.N., Deshmukh, R. and Deshmukh, S. (2014) ‘Performance Analysis of Different Sentence Oddity Measures Applied on Google and Google News Repository for Detection of Substitution’.

Dessì, D., Recupero, D.R. and Sack, H. (2021) ‘An assessment of deep learning models and word embeddings for toxicity detection within online textual comments’, *Electronics (Switzerland)*, 10(7). Available at: https://doi.org/10.3390/electronics10070779.

Devlin, J. *et al.* (2018) ‘BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding’, *NAACL HLT 2019 - 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies - Proceedings of the Conference*, 1, pp. 4171–4186. Available at: https://arxiv.org/abs/1810.04805v2 (Accessed: 3 April 2023).

Ding, B. *et al.* (2022) ‘Is GPT-3 a Good Data Annotator?’ Available at: http://arxiv.org/abs/2212.10450.

*facebook/bart-large-mnli · Hugging Face* (no date). Available at: https://huggingface.co/facebook/bart-large-mnli (Accessed: 9 August 2023).

Fong, S.W., Roussinov, D. and Skillicorn, D.B. (2008) ‘Detecting word substitutions in text’, in *IEEE Transactions on Knowledge and Data Engineering*, pp. 1067–1076. Available at: https://doi.org/10.1109/TKDE.2008.94.

Foote, K.D. (2023) *A Brief History of Natural Language Processing - DATAVERSITY*. Available at: https://www.dataversity.net/a-brief-history-of-natural-language-processing-nlp/ (Accessed: 6 August 2023).

Ghahramani, Z. (2023) *Google AI: What to know about the PaLM 2 large language model*. Available at: https://blog.google/technology/ai/google-palm-2-ai-large-language-model/ (Accessed: 29 July 2023).

*GPT - OpenAI API* (no date). Available at: https://platform.openai.com/docs/guides/gpt/chat-completions-api (Accessed: 15 July 2023).

*h2oGPT* (no date). Available at: https://gpt.h2o.ai/ (Accessed: 11 August 2023).

Hada, T. *et al.* (2021) ‘Codeword detection, focusing on differences in similar words between two corpora of microblogs’, *Annals of Emerging Technologies in Computing*, 5(2), pp. 90–102. Available at: https://doi.org/10.33166/AETiC.2021.02.008.

Harikrishnan N B (2019) *Confusion Matrix, Accuracy, Precision, Recall, F1 Score | by Harikrishnan N B | Analytics Vidhya | Medium*. Available at: https://medium.com/analytics-vidhya/confusion-matrix-accuracy-precision-recall-f1-score-ade299cf63cd (Accessed: 10 August 2023).

He, P. *et al.* (2020) ‘DeBERTa: Decoding-enhanced BERT with Disentangled Attention’, *ICLR 2021 - 9th International Conference on Learning Representations* [Preprint]. Available at: https://arxiv.org/abs/2006.03654v6 (Accessed: 9 August 2023).

Hosseini, H. *et al.* (2017) ‘Deceiving Google’s Perspective API Built for Detecting Toxic Comments’. Available at: https://arxiv.org/abs/1702.08138v1 (Accessed: 4 August 2023).

*Introduction - OpenAI API* (no date). Available at: https://platform.openai.com/docs/introduction/overview (Accessed: 3 April 2023).

*Introduction to Vertex AI  |  Google Cloud* (no date). Available at: https://cloud.google.com/vertex-ai/docs/start/introduction-unified-platform (Accessed: 29 July 2023).

Iyer, S. *et al.* (2022) ‘OPT-IML: Scaling Language Model Instruction Meta Learning through the Lens of Generalization’. Available at: http://arxiv.org/abs/2212.12017.

Joulin, A. *et al.* (2016) ‘FastText.zip: Compressing text classification models’. Available at: https://arxiv.org/abs/1612.03651v1 (Accessed: 30 July 2023).

Kaplan, J. *et al.* (2020) ‘Scaling Laws for Neural Language Models’. Available at: https://arxiv.org/abs/2001.08361v1 (Accessed: 3 April 2023).

Karabiber, F. (2023) *TF-IDF — Term Frequency-Inverse Document Frequency – LearnDataSci*. Available at: https://www.learndatasci.com/glossary/tf-idf-term-frequency-inverse-document-frequency/ (Accessed: 6 August 2023).

Khomsah, S., Ramadhani, R.D. and Wijaya, S. (2022) ‘The Accuracy Comparison Between Word2Vec and FastText On Sentiment Analysis of Hotel Reviews’, *Jurnal RESTI (Rekayasa Sistem dan Teknologi Informasi)*, 6(3), pp. 352–358. Available at: https://doi.org/10.29207/RESTI.V6I3.3711.

Lakshmanan Valliappa and Tigani Jordan (2019) *1. What Is Google BigQuery? - Google BigQuery: The Definitive Guide [Book]*. Available at: https://www.oreilly.com/library/view/google-bigquery-the/9781492044451/ch01.html (Accessed: 5 July 2023).

Leandro von Werra *et al.* (2023) *The Falcon has landed in the Hugging Face ecosystem*. Available at: https://huggingface.co/blog/falcon (Accessed: 11 August 2023).

Lewis, M. *et al.* (2019) ‘BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension’, *Proceedings of the Annual Meeting of the Association for Computational Linguistics*, pp. 7871–7880. Available at: https://doi.org/10.18653/v1/2020.acl-main.703.

Magu, R. and Luo, J. (2018) *Determining Code Words in Euphemistic Hate Speech Using Word Embedding Networks*.

Mehandzhiyski, V. (2023) *What Is an Autoregressive Model? | 365 Data Science*. Available at: https://365datascience.com/tutorials/time-series-analysis-tutorials/autoregressive-model/ (Accessed: 9 August 2023).

Mikolov, T. *et al.* (2013) ‘Efficient Estimation of Word Representations in Vector Space’, *arXiv: Computation and Language* [Preprint].

Moore, J. (2022) *Hidden in plain sight: How the dark web is spilling onto social media | WeLiveSecurity*. Available at: https://www.welivesecurity.com/2022/02/10/hidden-plain-sight-dark-web-spilling-social-media/ (Accessed: 2 April 2023).

*MoritzLaurer/DeBERTa-v3-base-mnli-fever-anli · Hugging Face* (no date). Available at: https://huggingface.co/MoritzLaurer/DeBERTa-v3-base-mnli-fever-anli (Accessed: 9 August 2023).

*MPT-30B: Raising the bar for open-source foundation models* (no date). Available at: https://www.mosaicml.com/blog/mpt-30b (Accessed: 11 August 2023).

*multi\_nli · Datasets at Hugging Face* (no date). Available at: https://huggingface.co/datasets/multi\_nli (Accessed: 9 August 2023).

Mwiti, D. (2021) *Guide to Using Pre-trained Word Embeddings in NLP*. Available at: https://blog.paperspace.com/pre-trained-word-embeddings-natural-language-processing/ (Accessed: 8 August 2023).

OpenAI (2023) *Introduction - OpenAI API*. Available at: https://platform.openai.com/docs/introduction/overview (Accessed: 15 July 2023).

*PaLM 2 for Text – Vertex AI – ResearchProposal – Google Cloud console* (no date). Available at: https://console.cloud.google.com/vertex-ai/publishers/google/model-garden/text-bison?project=researchproposal (Accessed: 29 July 2023).

Pennington, J., Socher, R. and Manning, C.D. (2014) ‘GloVe: Global Vectors for Word Representation’, in *Empirical Methods in Natural Language Processing (EMNLP)*, pp. 1532–1543. Available at: http://www.aclweb.org/anthology/D14-1162.

Pfaff, K.L., Gibbs, R.W. and Johnson, M.D. (1997) ‘Metaphor in using and understanding euphemism and dysphemism’, *Applied Psycholinguistics*, 18(1), pp. 59–83. Available at: https://doi.org/10.1017/S0142716400009875.

*Quotas and limits  |  Vertex AI  |  Google Cloud* (no date). Available at: https://cloud.google.com/vertex-ai/docs/quotas (Accessed: 29 July 2023).

Raffel, C. *et al.* (2019) ‘Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer’, *Journal of Machine Learning Research*, 21, pp. 1–67. Available at: https://arxiv.org/abs/1910.10683v3 (Accessed: 9 August 2023).

Reynolds, L. and McDonell, K. (2021) ‘Prompt Programming for Large Language Models: Beyond the Few-Shot Paradigm’. Available at: http://arxiv.org/abs/2102.07350.

S. Dixon (2023) *Number of worldwide social network users 2027 | Statista*. Available at: https://www.statista.com/statistics/278414/number-of-worldwide-social-network-users/ (Accessed: 2 April 2023).

SHARMA, S. (2017) *Epoch vs Batch Size vs Iterations | by SAGAR SHARMA | Towards Data Science*. Available at: https://towardsdatascience.com/epoch-vs-iterations-vs-batch-size-4dfb9c7ce9c9 (Accessed: 10 August 2023).

Shung, K.P. (2018) *Accuracy, Precision, Recall or F1? | by Koo Ping Shung | Towards Data Science*. Available at: https://towardsdatascience.com/accuracy-precision-recall-or-f1-331fb37c5cb9 (Accessed: 10 August 2023).

Skillicorn, D.B. (2005) *Beyond Keyword Filtering for Message and Conversation Detection*. Available at: www.fabrica.it/.

Tassone, J. *et al.* (2020) ‘Utilizing Deep Learning to Identify Drug Use on Twitter Data.’, *arXiv: Social and Information Networks* [Preprint].

*The Crime and Terror Threat on Social Media — ACCO* (no date). Available at: https://www.counteringcrime.org/crime-and-terror-threat-on-social-media-fact-sheet (Accessed: 4 August 2023).

*The Stanford NLI Corpus Revisited - The Stanford Natural Language Processing Group* (no date). Available at: https://nlp.stanford.edu/blog/the-stanford-nli-corpus-revisited/ (Accessed: 9 August 2023).

The Vicuna Team (2023) *Vicuna: An Open-Source Chatbot Impressing GPT-4 with 90%\* ChatGPT Quality | LMSYS Org*. Available at: https://lmsys.org/blog/2023-03-30-vicuna/ (Accessed: 11 August 2023).

Tom Allard, P.W. (2020) *Asia-Pacific drug trade thrives amid the COVID-19 pandemic | Reuters*. Available at: https://www.reuters.com/article/us-asia-crime-drugs/asia-pacific-drugtradethrives-amid-the-covid-19-pandemic-idUSKBN22R0E0 (Accessed: 2 April 2023).

*Tune language foundation models  |  Vertex AI  |  Google Cloud* (no date). Available at: https://cloud.google.com/vertex-ai/docs/generative-ai/models/tune-models (Accessed: 11 August 2023).

Vaswani, A. *et al.* (2017) ‘Attention Is All You Need’, *Advances in Neural Information Processing Systems*, 2017-December, pp. 5999–6009. Available at: https://arxiv.org/abs/1706.03762v5 (Accessed: 3 April 2023).

Wei, J. *et al.* (2021) ‘Finetuned Language Models Are Zero-Shot Learners’, *ICLR 2022 - 10th International Conference on Learning Representations* [Preprint]. Available at: https://arxiv.org/abs/2109.01652v5 (Accessed: 9 August 2023).

Wei, J., Wang, X., *et al.* (2022) ‘Chain-of-Thought Prompting Elicits Reasoning in Large Language Models’. Available at: http://arxiv.org/abs/2201.11903 (Accessed: 9 August 2023).

Wei, J., Tay, Y., *et al.* (2022) ‘Emergent Abilities of Large Language Models’. Available at: https://arxiv.org/abs/2206.07682v2 (Accessed: 9 August 2023).

*Word embeddings in NLP: A Complete Guide* (no date). Available at: https://www.turing.com/kb/guide-on-word-embeddings-in-nlp (Accessed: 6 August 2023).

Yin, W., Hay, J. and Roth, D. (2019) ‘Benchmarking Zero-shot Text Classification: Datasets, Evaluation and Entailment Approach’, *EMNLP-IJCNLP 2019 - 2019 Conference on Empirical Methods in Natural Language Processing and 9th International Joint Conference on Natural Language Processing, Proceedings of the Conference*, pp. 3914–3923. Available at: https://doi.org/10.18653/v1/d19-1404.

Yuan, K. *et al.* (2018) ‘Reading Thieves’ Cant: Automatically Identifying and Understanding Dark Jargons from Cybercrime Marketplaces’, in *USENIX Security Symposium*.

van der Zee, Y. *et al.* (2021) ‘Code Word Detection in Fraud Investigations using a Deep-Learning Approach’. Available at: http://arxiv.org/abs/2103.09606.

Zhao, W.X. *et al.* (2023) ‘A Survey of Large Language Models’. Available at: http://arxiv.org/abs/2303.18223.

Zhu, W. *et al.* (2021) ‘Self-supervised euphemism detection and identification for content moderation’, in *Proceedings - IEEE Symposium on Security and Privacy*. Institute of Electrical and Electronics Engineers Inc., pp. 229–246. Available at: https://doi.org/10.1109/SP40001.2021.00075.

Zulkifli, H. (2018) *Understanding Learning Rates and How It Improves Performance in Deep Learning | by Hafidz Zulkifli | Towards Data Science*. Available at: https://towardsdatascience.com/understanding-learning-rates-and-how-it-improves-performance-in-deep-learning-d0d4059c1c10 (Accessed: 10 August 2023).

# Appendix

**Technical Specifications:**

**Hardware:**

OS: Microsoft Windows 11

CPU: 11th Gen Intel i7 (4 cores)

GPU: Not used for this research

**Software Versions and Packages Used:**

Python version – 3.9.12

IDE: Visual Studio- 1.81

Notebook: Google Collab, Jupyter Notebook

**Packages used:**

* FastText
* Pandas
* sklearn.metrics
* Os
* Pathlib
* Re
* seaborn
* matplotlib.pyplot

**APIs Used:**

* OpenAI API for GPT-3
* Vertex AI API for PaLM
* Hugging Face Inference API for BART and DeBERTa
* BigQuery API for preparing dataset

**Reference Manager:** Mendeley Reference Manager