EAI Endorsed Transactions
on AI and Robotics

Research Article EAI.EU

A Study Towards Building Content Aware Models in NLP using Genetic Algorithms

Umesh Tank[1], Saranya Arirangan[2], Anwesh P R[2, *] and Narayana Darapaneni[3]

[1] PES University, Bangalore, Karnataka, 560050, India
[2]Great Learning, Hyderabad, Telangana, 500089, India
[3]Northwestern University, Evanston, IL 60208, United States

Abstract

INTRODUCTION: With the advancement in the large language models, often called LLMs, there has been increasing concerns around the usage of these models. As they can generate human-like text and can also perform a number of tasks such as generating code, question answering, essay writing and even generating text for research papers.
OBJECTIVES: The generated text is subject to the usage of the original data (using which models are trained) which might be protected or may be personal/private data. The detailed description of such concerns and various potential solutions is discussed in 'Generative language models and automated influence operations: Emerging threats and potential mitigations'.
METHODS: Addressing these concerns becomes the paramount for LLMs usability. There are several directions explored by the researchers and one of the interesting works is around building content aware models. The idea is that the model is aware of the type of content it is learning from and aware what type of content should be used to generate a response to a specific query.
RESULTS: In our work we explored direction by applying poisoning techniques to contaminate data and then applying genetic algorithms to extract the non-poisoned content from the poisoned content that can generate a good response when paraphrased.
CONCLUSION: While we demonstrated the idea using poisoning techniques and tried to make the model aware of copyrighted content, the same can be extended to detect other types of contents or any other use cases where content awareness is required.

Keywords: Content awareness, Large Language Models, Data poisoning, Genetic algorithms

Received on 05 October 2023, accepted on 03 November 2023, published on 07 November 2023

Copyright © 2023 U. Tank *et al.*, licensed to EAI. This is an open access article distributed under the terms of the CC BY-NC-SA 4.0, which permits copying, redistributing, remixing, transformation, and building upon the material in any medium so long as the original work is properly cited.

doi: 10.4108/airo.4078

[*]Corresponding author. Email: anwesh@greatlearning.in

1. Introduction

1.1. The Imitation Game

The history of the NLP and intelligence dates back to at least 1950 when Alan Turing proposed a famous Turing test. Since then, researchers have been trying to build a system that can impersonate humans and more recently the focus is to build a useful AI system that can automate some of the jobs that require conversation with the human in real-time and more often in the open domain setting. These systems are mostly known as a Chatbot.

Chatbots are computer programs that possess the ability to engage in natural language conversations with users. They have the capacity to comprehend user intentions and provide responses using pre-established rules and data. Chatbots are specifically crafted to emulate the conversational behavior of humans. The capabilities of the Chatbots have been extended since their inception and now they don't just stick to the said definition but are able to take images and many other forms of inputs to perform a given task.

1.2. Power comes with great responsibilities

As Chatbots become more and more powerful, comes the concerns around their handling of information, specifically copyrighted content, sensitive/confidential information, and audience specific information. Chatbots, like ChatGPT, inadvertently use that information to generate a response to a user query. This requires models to be smart enough to generate user appropriate responses just like humans do. We don't disclose classified/sensitive information inadvertently, we use copyrighted content within the permissible limits. This is where **content awareness** becomes the most important when it comes to information handling.

1.3. Elephant in the room

In our work, we present a study of building a content-aware model using genetic algorithms. The problem statement we are trying to solve is - How can we build content awareness into the model so that they can handle information just like humans? To answer this question, we conducted a study and developed an approach where we apply data poisoning techniques to copyrighted content and contaminate it. Once we have a dataset that includes poisoned and non-poisoned content our next objective is to extract contents from the corpus such that the model can generate meaningful paraphrased sentences. Extracting an optimal subset of content from the corpus requires us to find a solution in the huge search space. This is where genetic algorithms excel [6] & [7]. Thus, we apply genetic algorithms to it to extract the best candidate solution. Note that the genetic algorithm falls under the umbrella term called evolutionary algorithms. This is our first attempt towards building content aware models.

2. Related Work

2.1. Information handling

Until recently, handling copyrighted or confidential information was mostly ignored because it is a sensitive matter and involves litigations. Thus, the focus was on building more powerful models. But with the success of LLMs came the challenges of handling this information which made clear that these issues are real and need to be addressed [13] & [15]. The way information is handled by the models and the associated risk and misuse of that information is an active research area. There are several contributions from researchers and organizations [14] & [16]. One of the most notable and detailed works we came across is captured in [1]. As described in [1] it is not hard to use LLMS to generate misinformation. This can lead to numerous issues such as generating toxicity, getting biased opinions, spreading fake news, or building hostility. Various online platforms have been misused to influence the audience and LLMs can't hide for a long time [17]. On the other hand, if copyright contents are not handled cautiously then it can lead to litigations.

The content-aware AI systems can be rescued for issues related to information handling. The content-aware AI systems are capable of recognizing misinformation and preventing fake information from getting generated by LLMs. The content awareness also enables AI system to detect copyright content and exclude them from the response.

2.2. Current research

Since LLMs are available very recently, issues related to information handling are also new and being investigated. There isn't any method available at this moment to the best of our knowledge which effectively solves information handling issues. When we asked one of the famous chatbots if its generated response could infringe copyright. The response we received was that it is the responsibility of the users to ensure that information received from the generated response wouldn't violate copyright.

Although methods don't exist to prevent copyright contents from getting generated by the LLMs, it's an active research area and various techniques do exist, such as fingerprinting or watermarking the generated output [18], Building Content Aware Models, building models that produce detectable output (eg using radioactive data) as described in [19]. To detect generated output some of the techniques explored are Natural Language Steganography [20], Simulated Annealing, radioactive data [19], and Data Poisoning [10]. In our study, we used data poisoning techniques and genetic algorithms to develop a model.

2.**3**. Data Poisoning Techniques

Extensive research has shown that ML models are vulnerable to adversarial attacks [8]. So far visual models are the primary targets of such attacks. The NLP models are no exception, and they are also vulnerable to such attacks [9]. The NLP models can be affected in black-box settings without making any perceptible changes to the data/text. The techniques are often called Data Poisoning in which various perturbation methods are used which are not visible to human eyes [11] & [12]. Research has shown [3] that most of the models are not immune to Data Poisoning attacks [10]. Moreover, Data Poisoning attacks can be carried out in a targeted manner without any understanding of the model. The following methods can be used or they can be combined with other methods to generate more sophisticated changes to the input.

1. Invisible Character
2. Homoglyph
3. Reordering

We will describe the above methods briefly in the following sections. More details can be found in this paper [3]. Though these methods are imperceptible to humans, they do modify inputs that can be detected by the machine.

Invisible Characters

How many characters are there in the below sentence? " invisible characters." A human can see 21 characters but a machine will see more than 21. You can copy the above text and calculate its length using Python's len function and you would notice that it returns 27. Therefore the above text contains 6 invisible characters. The invisible characters are zero-width characters that are ignored by most of the text rendering systems but are visible to machines. You can find more information about the zero-width characters in the original paper[3]. In our work, we just considered zero-width space, zero-width joiner, and zero-width non-joiner.

Homoglyph

Homoglyph refers to the characters that look similar but have different meanings. They are two distinct characters but have similar glyphs. For example the letter 'O' and zero (0). Similarly, the Latin letter 'H' and Cyrillic letter 'H' also capitalized i (I) and l.

Reordering

While most of the scripts are read from left to right there are scripts that are written from right to left. Therefore, there are directionality control characters that control the direction of the script. For example, - '001' and 'U+202E100' are the same. Moreover, you wouldn't be able to see 'U+202E' but just 100 but since 'U+202E' direction control character will revert the direction for the system it means the same.

2.4. Genetic Algorithms

The genetic algorithms are part of the evolutionary algorithms which are inspired by the theory of evolution, described by Charles Darwin. The genetic algorithm is a widely used branch of evolutionary algorithms while other algorithms exist. In this section, we will describe genetic algorithm's resemblance to evolution, their fundamental operations, their typical workflow, and various steps that need to be implemented.

What are genetic algorithms?

Genetic algorithms, which draw inspiration from natural evolution, form a group of search algorithms. By simulating natural selection and reproduction, these algorithms generate excellent solutions for diverse problems in search, optimization, and learning. Moreover, the resemblance to natural evolution enables genetic algorithms to surmount challenges faced by conventional search and optimization algorithms, particularly when dealing with problems characterized by numerous parameters and intricate mathematical representations.

For natural selection and reproduction, Genetic algorithms utilize a simplified version of the evolutionary processes observed in nature, as described by Darwin. The fundamental principles of Darwinian evolution can be summarized as follows:

1. Variation (Principle of Variation): In every species, individuals possess unique genetic structures, resulting in a multitude of distinct variations in their physical characteristics. Within a population, individuals differ from one another and display a diverse range of traits or attributes.
2. Inheritance (Principle of Inheritance): Individuals transmit a portion of their genetic material to their offspring, resulting in the inheritance of traits from parents to their offspring. Some of these traits are consistently passed down from the parents because of that offspring tend to resemble their parents more closely than the species.
3. Selection (Principle of Selection): Certain individuals possess inherited traits (genes) that provide them with an advantage in surviving within a competitive environment or increasing their reproductive success. As a result, their offspring are more likely to thrive in the same competitive environment and produce their own offspring. Consequently, the prevalence of their genes increases throughout the entire population, as certain variants reproduce more frequently than others.

Essentially, evolution sustains a population of specimens that possess varying characteristics. Those specimens that are better adapted to their environment have higher chances of survival, reproduction, and passing on their traits to subsequent generations. Over time, this process leads to species becoming increasingly adapted to their environment and the challenges they encounter.

Each individual in the population is a potential solution therefore population is a solution search space where our objective is to find the optimal solution.

In the genetic algorithm, we need to iterate over many generations. We need to find fitter individuals from the population which will represent the next generation. To evaluate individuals, a genetic algorithm uses a fitness function which computes the fitness of an individual in the population. In other words, the fitness function can compare two individuals from the search space and identify a better one. The fitness function can be a score or a simple comparison of two individuals. Individuals who attain higher fitness scores signify superior solutions and are more inclined to be selected for reproduction, subsequently contributing to the composition of the next generation.

Genetic algorithm's Operator

Crossover or recombination, where offspring inherit a mixture of their parents' traits, plays a crucial role in enabling evolution. Crossover helps maintain population diversity and gradually brings together favorable traits. Additionally, mutations, which are random variations in traits, can occasionally introduce changes that result in significant advancements. The three primary operators of genetic algorithm are briefly described below.

1. Selection
2. Crossover
3. Mutation

1) Selection

Once the fitness of everyone within the population has been computed, a selection procedure is employed to determine which individuals will have the opportunity to reproduce and generate the offspring that will constitute the subsequent generation.

This selection process is primarily guided by the fitness scores assigned to the individuals. Those with higher scores are given a greater likelihood of being selected and transmitting their genetic material to the next generation.

Although individuals with lower fitness values can still be chosen, their probability of selection is comparatively lower. This approach ensures that their genetic material is not entirely disregarded, allowing for a degree of inclusion in the reproductive process.

2) Crossover

During the evolution process, new offspring are generated by combining two individuals from the current generation, also called parents. Parents will exchange or crossover their chromosomes to create new chromosomes for the offspring. Since each parent has a chromosome, usually crossover process ends up creating two offspring.

3) Mutation

The mutation alters the existing chromosome of an individual thereby introducing new traits and attributes to the individual. The mutation helps in changing attributes which allows the algorithm to explore uncharted solution space. The mutation operator is applied randomly and periodically. A mutation may be in the form of a random change in a single gene or set of genes in the chromosomes.

Genetic algorithm's Workflow

The typical workflow of genetic algorithm is straightforward. It iterates over the population while evolving during each iteration by applying operators. The goal of evolution is to retain the fitter individuals in the population.

The Fig-1 below illustrates the typical workflow of a genetic algorithm.

The below steps outline the workflow.

1. Define initial population.
2. Compute the fitness of everyone in the population using a fitness function.
3. Select N individuals from the population that will represent the current generation.
4. Perform crossover to get the next generation.
5. Apply mutation on the new generation.
6. Compute the fitness of everyone in the new generation.
7. Check if we reached the stopping condition.
8. If the stopping conditions aren't met go back to #3
9. If the stopping conditions are met, then select the best N individuals that represent the solution.

In summary, the genetic algorithm process initiates with a population of candidate solutions (individuals) that are randomly generated. These individuals are then assessed using the fitness function. The main procedure involves a loop where selection, crossover, and mutation operators are sequentially applied to the individuals, followed by their re-evaluation. This loop persists until a predefined stopping condition is satisfied, at which point the best individual within the population is chosen as the solution.

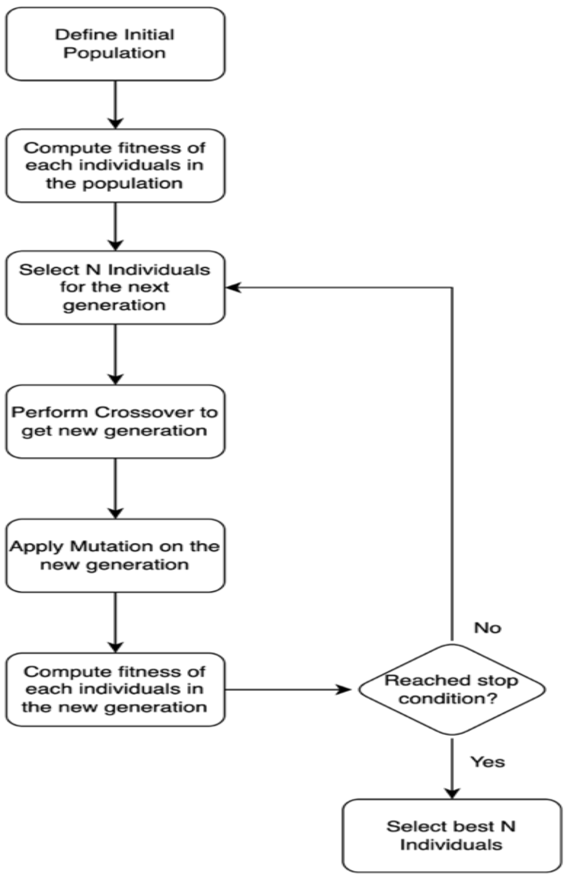
Define Initial Population
Compute fitness of each individuals in the population
Select N Individuals for the next generation
Perform Crossover to get new generation
Apply Mutation on the new generation
Compute fitness of each individuals in the new generation
Reached stop condition?
No
Yes
Select best N Individuals

Figure. 1: Genetic Algorithm Workflow

3. Materials and Methods

We followed a standard machine learning pipeline that is Data Generation, Genetic Algorithm Flow, and Result Analysis. The Data Generation includes Data collection, Data cleaning, and Data preparation. The GA flow covers the workflow of Genetic Algorithms (such as population selection, Crossover, Mutation, and stopping condition). The Result Analysis covers generating results, computing evaluation metrics, and deriving the conclusion.

3.1. Dataset

Since our primary goal is to generate a text from the given input, we need to find a dataset that can be used for the text generation. We have used Microsoft Research Paraphrase

Corpus (https://www.microsoft.com/en-us/download/details.aspx?id=52398) as our data set. The dataset is suitable for paraphrasing; hence we can easily paraphrase the given input. There were a few other datasets that we considered but due to GPU constraint, we thought of using only the subset of Microsoft Research Paraphrase Corpus. The other datasets we considered are Quora Question Pairs Datasets and Google PAWS-wiki (Paraphrase Adversaries from Word Scrambling).

3.2. Architectural overview/Proposed Architecture

The overall high-level architecture of our proposed solution is depicted in the following diagram, Fig-2. The usual components involved in the generative models are train data, prepared during the data generation phase, which will be fed to transformers to learn from the data. Once the model is trained on the dataset we deploy it, and the transformer produces a response based on the user query. The text generator module selects the appropriate response based on the context and other user-specific parameters. We have introduced a few modules in the typical flow of generative models which are Data Poisoning, GA module, and filtered contents.

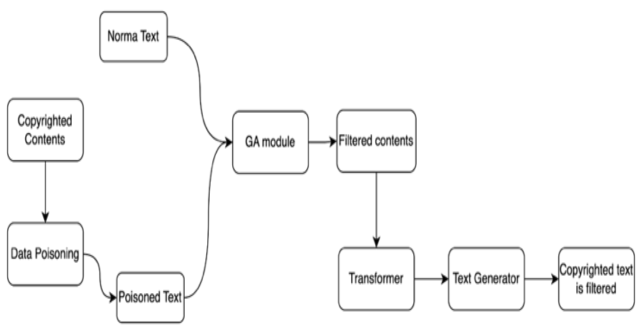

Figure. 2: Proposed System Architecture

In our work, to demonstrate our technique of filtering out copyrighted content, we have used Microsoft Research Paraphrase Corpus. As part of the data collection, we applied data poisoning to the dataset by injecting zero-width invisible characters into the original text. The invisible characters are zero-width characters that are ignored by most of the text rendering systems but are visible to machines. In the next section, we have covered data poisoning techniques in a little more detail. In our work, we just considered zero-width space, zero-width joiner, and zero-width non-joiner. We poisoned 50% of the characters distributed uniformly to ensure noise was high in the dataset.

For our experiment, we would like to mimic a situation where a dataset contains copyrighted content, and we are differentiating it using poisoned data. Therefore, we need to mix normal text with the poisoned data. We picked the first 400 lines from the normal text and 400 lines from the poisoned dataset and put all the sentences together in a single data set.

Each sentence will be presented in the chromosomes, and the sentence can be visualized as a gene in the chromosomes. For our representation, a gene is either present or absent. We shuffled the dataset so that poisoned and original contents were mixed and not in sequence. Though it is not a requirement, it is good to have some randomness.

In general algorithms don't work on text directly but work on the numeric form of the text. Therefore, we need to encode or represent text in the numeric format. In a format that is suitable for the algorithm. One of the challenges with the Genetic algorithm is presenting the solution space in the numeric format such that we can apply various operators of the algorithms, e.g., fitness functions, crossover, and mutation.

3.3. Data coding and presentation

We found the perplexity score would be the best fit for our use case, since the perplexity score is a float number, and each sentence can have different perplexity values. It would allow us to compare the fitness of the two genes in finding the better solution.

We decided to use the average perplexity value of individuals for fitness calculation. The individuals or chromosomes are represented by a series of 0 or 1. Which indicates if a particular gene is present (1) or absent (0). Each sentence in the dataset is represented as a gene. Thus, to represent an individual or chromosome we will need 800 genes as our dataset consists of 800 lines of text.

The perplexity score was calculated based on the model that we had built which is based on Sequence-to-sequence [4] transformer [2] from the Huggingface and was pretrained by Facebook. For each sentence 3 sentences were paraphrased, and its perplexity score was calculated. Since our dataset is not dynamic and we don't plan to change it during our experiment, we computed the score for all the sentences in the corpus once and stored it in a file along with the original sentence for which the score was computed.

3.4. Data Poisoning

As described in the previous section, we applied data poisoning techniques on the dataset to contaminate and assume that it is copyrighted content which we would like the model to identify and filter out. Almost 50% of the data was poisoned by adding invisible characters. There were 3 zero-width characters that we have used: zero-width space (0x200B), zero-width joiner (0x200D), zero-width non-joiner (0x200C). There are other zero-width characters as well as other ways of manipulating data but for our work, we are limiting ourselves to invisible zero-width characters.

To inject zero-width characters we have used uniform distribution to identify an index where we would like to inject the zero-width character.

3.5. System Requirements

For our experiments, we needed a model that can generate a text. Therefore, we built and trained a transformer-based model for the text generation and computed the perplexity score. Instead of reinventing the wheel, we used existing BART based model developed by Facebook. Here are some details about the model-

- **Algorithms:** Xformers, Huggingface transformer
- **Model:** Facebook bart-base [21]
- **Dataset:** Microsoft Research Paraphrase Corpus (https://www.microsoft.com/en-us/download/details.aspx?id=52398)
- **Techniques:** Imperceptible perturbation for data poisoning which is zero-width non-joiner invisible character in the input text.
- **Model Type:** Text generation
- **Platform:** Google colab

Training this model on a data set requires GPU hence we used a free version of google colab which has GPUs. Since it is a free version it has certain resource constraints, due to which we just ran 2 epochs. Although there are more powerful BART base models developed by Facebook which can be employed if resources are not constrained.

Another algorithm we developed was the genetic algorithm. The output from the text generation model is a set of text which includes both poisoned and non-poisoned data. The job of the algorithm is to find the set of text that can generate a response that filter out the copyrighted text. We have used our own machines to run genetic algorithms on the dataset. We have used a small subset of the data to get results in a reasonable amount of time and resources. Though we used a small subset of the data, we did try different thresholds for the stopping condition, up to an extent where we could see significant improvements. You can find our work in the following repo- https://github.com/umeshgtank/content_awareness_study

3.6. GA Module

In the GA module, we apply genetic algorithms to the prepared data. The final dataset includes the original text and its perplexity score. We will use this perplexity score to evaluate everyone in the population. We need to define certain parameters for the genetic algorithms and need to set some initial values for the hyperparameters. Below is a list of hyperparameters along with the values we used during our experiments.

- Population size: **10000**
- Number of chromosomes: **800**
- Selection method: **Tournament selection with tournament size 3**
- Fitness function: Average perplexity score. **Minimize the average perplexity score** for the individuals.
- Crossover method: **Two-point crossover**. Cross two individuals with a probability 0.5
- Mutation method: **flip bit** method with 0.05 chromosome mutation probability and 0.2 for mutation of individual
- Termination conditions:
  - The average perplexity **score is < 4.**
  - 1000 Generation

As we have seen in the previous sections, the workflow of Genetic Algorithms is relatively simple and straightforward to implement but Genetic Algorithms have quite a few operators, such as selection, crossover, mutation, etc. As noted in the previous section, there are multiple algorithms and methods for each of these operators. Each of the methods has its own characteristics and produces a different result or can be applied to a specific use case. Since implementing all of those methods and algorithms is challenging, time-consuming and, error-prone, it is a good idea to leverage a proven framework which will allow us to focus on our core research and try out different scenarios instead of worrying about the implementation of those operators.

This is where theDEAP framework fills the gap. The DEAP (Distributed Evolutionary Algorithm in Python) framework as its name suggests is an open-source Python-based development framework focused on evolutionary algorithms. The DEAP framework implements data structures and algorithms for genetic algorithms, thus we implemented our workflow using the DEAP framework.

Let’s look at how we have implemented the GA workflow:

Defining Initial Population

As we have noted in the previous section, Genetic Algorithms start with the initial population and search for the solution in the initial population. The population is defined as a chromosome and a chromosome represents the candidate solution; therefore, the population represents the current generation or current state of the solutions in the search space. That is one chromosome is one of the candidate solutions in the search space. If there are N chromosomes (that is N individuals in the population) then there are N candidate solutions. The initial population is a hyperparameter and that needs to be tuned. The bigger the population, the more varieties we can get in each generation and can represent more solutions from the search space. Therefore, having the right size of population increases our chance to find an optimal solution. Below is how the initial population can be defined using the DEAP framework.

```
toolbox = base.Toolbox()
toolbox.register("individual",
                 tools.initRepeat,
                 creator.Individual,
                 toolbox.attr_bool,
                 800)
```

Gene

Chromosomes are represented as a sequence of a gene, very similar to a biological system. A gene can be defined as a sequence of binary digits, integers, or real numbers depending on the problem at hand. The gene represents the contribution of individual data points to the solution. Changing the value of a gene changes the solution and it represents a different solution than before. The main objective of Genetic Algorithms is to find the values for the set of genes (called chromosomes) which is the best solution in the given search space.

As discussed in the problem statement section, we would like to find a set of sentences from a corpus that can generate better responses based on the perplexity score. Therefore, each sentence can be represented as a gene. We decided to use a binary string to represent each sentence in the corpus as a gene. Thus, genes can be present or absent in the chromosome based on the binary value assigned to it. As our dataset consists of 800 lines of text we have 800 genes in one chromosome or in other words an individual consists of 800 genes.

The individual is a vector representation of a binary digit, and the population can be a set of a vector. Thus, the population can be represented as a matrix of individuals and genes. The initial population can be visualized as shown in the below Table-1. Each column in the table represents a sentence in the corpus. Each row in the table represents an individual or chromosome. The value in each column of the individual's row represents whether the given sentence is present or absent. The entire row is a vector representation of a single individual whose characteristic contributes to a solution (non-poisoned sentences). The entire table is our population, and it is a matrix of Boolean values.

Table-1: Population Presentation

	Those reports were denied by the interior minister, Prince Nayef.	The year-ago comparisons were restated to include Compaq results.	It was the best advance since Oct. 1, when the index gained 22.25.	….	Ricky Clemons' brief, troubled Missouri basketball career is over.
Ind-1	0	1	0	…	1
Ind-2	0	0	1	…	1
Ind-3	1	1	0	…	0
…..					
Ind-N	1	0	1		0

To define initial the population we uniformly sampled sentences from the corpus and assigned them to an individual. So, there is a 50% chance that a sentence will be present in the individual. Using this approach N individuals are generated to create the initial population. Below is how we defined genes using the DEAP framework.

```
toolbox.register("attr_bool",
                 random.randint,
                 0,
                 1)
```

Fitness computation

Fitness is a function that we would like to optimize. At each iteration of the Genetic Algorithm's workflow individuals are evaluated using the fitness function. Fitness helps formulate the problem statement using which GA tries to find the solution.

Since we are using a paraphrase dataset, we found that a perplexity score is a better way of measuring the quality of paraphrased sentences given an input sentence. Thus, the perplexity score is applied to a gene and not the entire chromosome. Therefore, we take an average perplexity score for genes which is the fitness value for each individual. Here is how we implemented the fitness function. Note that we have computed the perplexity score upfront to avoid doing it for every single comparison.

```
def sol_fitness(individual):
  indices = [i for i, x in enumerate(individual) if (x == 1)]
  ind_fitness_score = get_avg_score(indices)
  return ind_fitness_score,
```

Selecting Individuals from the Population

Once we computed the fitness score for everyone, we needed to select the fittest individuals who would represent the next generation. There are several methods available to select an individual from the population. For our use case, tournament selection is best suited as we would like to select the fittest individuals. In our algorithm, we randomly select 3 individuals and out of those we select the fittest individual which will represent the next generation. The above process is repeated to select everyone in the population. The size of the population and selecting the fittest individual from the randomly chosen individuals are hyperparameters and it can be optimized for the given use case. Below code snippet below demonstrates the tournament selection.

```
toolbox.register("select",
                 tools.selTournament,
                 tournsize=3)
```

Crossover

We employed a two-point crossover method on the individuals to get offspring. In the two-point crossover, we will identify two points in the sequence and all the genes between these two points are swapped with each other. For example, if two points are 2 and 4 then all the genes between 2-4 are exchanged between two selected individuals.

After crossover, the average value of the genes will change which means the fitness of the individual will be changed as well. During each iteration, we applied crossover on 50% of the population. The crossover points were randomly selected for each crossover operation. Meaning crossover points selected for one pair of chromosomes may not be selected for the next pair of chromosomes but will be selected randomly. Below code block below defines a two-point crossover

```
    toolbox.register("crossover",
                      tools.cxTwoPoint)
```

Mutation

Like Selection and Crossover, there are several methods to apply mutation. Some of these mutation methods and methods of other genetic operators are described in more detail in [5]. Since we are representing individuals as binary string Flip bit mutation fit perfectly well for our use case.

As we can see a gene is chosen randomly and its value is flipped which impacts the representation of that gene in the individual that eventually impacts the fitness score of that individual.

We mutated around 20% of the individuals from the population during each cycle and mutated only 2% of the genes from the selected individuals. Since each individual consists of 800 genes in our use case, we flipped roughly 16 genes of selected individuals during each cycle. Below code snippet implements the same

```
    toolbox.register("mutate",
                     tools.mutFlipBit, indpb=0.05)
```

Stopping Condition

Genetic Algorithms can run for a long time or even forever since the idea is to iterate over the existing solution in search of a better solution and the final solution is not known. Therefore, defining a stopping condition is critical to stop the algorithm when it finds the optimal solution. In most cases, the stopping condition is associated with the fitness function as that is the function that we would like to optimize hence the stopping condition can be some value of that fitness function. During our experiments, we discovered that the perplexity score for the sentences that perform better is mostly <2. Therefore, our stopping condition is when the average score of the individual is less than 2. It is likely that after many iterations and producing many generations algorithm may not be able to meet the condition, so to avoid GA running forever it is a good idea to add an additional condition that breaks after producing a certain number of generations. We will stop our GA after 1000 generations. The stopping condition can also sometimes be a hyperparameter. To get better results we may want to set strict values and it can be relaxed to reduce the computation cost.

4. Results

Since GA can be computationally expensive, we tried a corpus with 800 sentences which is a mix of poisoned and non-poisoned data. We would like to filter out poisoned data so that we can have a dataset that can generate better results for the given query. We can see the fitness of the solution increases when more generations are produced. In our case, better results are those where the dataset has a smaller number of poisoned contents. We tried to bring down the average fitness score to < 2 which resulted in producing 146 generations with just 800 lines of content. Fig. 3 below demonstrates fitness improvement along with the increase in a number of generations produced. We also tried to visualize the average population evaluated in the same graph. When the average perplexity score is <2, we can see produced results are promising. We can see generated text does have some poisoned text but despite that overall dataset is able to generate a better response compared to the one trained on the mixed dataset.

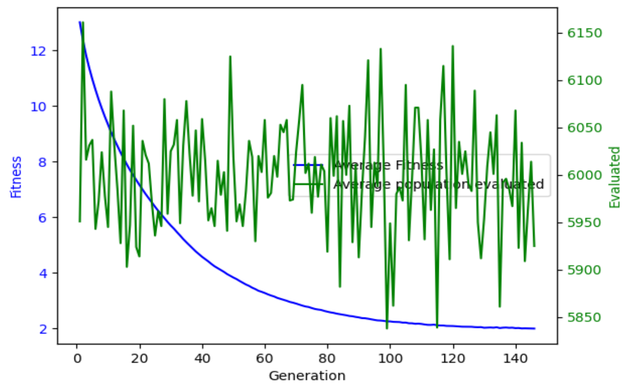

Figure. 3: Fitness (<2) vs Generation

The above chart illustrates the average fitness score over a generation. Note that we would like to minimize the perplexity score thus lower value indicates fitter individuals.

We can set a stricter value for the average perplexity score to reduce the noise (poisoned data) in the results. We tried different values (Fig.4) and discovered that decreasing the average perplexity score improves the results significantly at the same time it also needs to produce more generations.

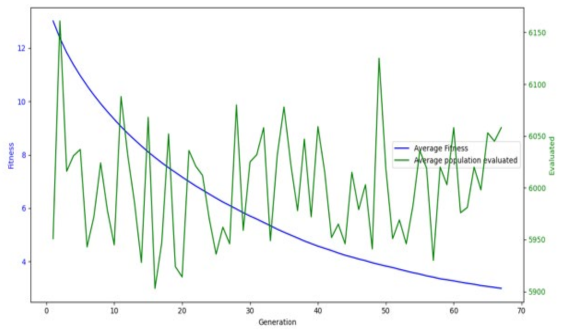

Figure. 4: Fitness (<3) vs Generation

Finding the ideal value for the stopping condition can be challenging but if found it produces remarkably efficient results. As we can see in the below image, when we reduce the fitness threshold, we can see algorithm produces more generations and there are a less number of poisoned contents in the output. When we increase the threshold, we can see more poisoned contents are generated. It is worth noting that when we reduce the threshold non-poisoned data is not impacted much. This mean when a genetic algorithm produces more generation, the population evolves by producing fitter individuals, and we get closer to the optimum solution. Remember, we are using a perplexity score hence we need to reduce the threshold to get better results as the lower the perplexity scores better the result. The below column chart (fig.5) shows the number of generations, filtering of poison & non-poisoned data, and fitness threshold.

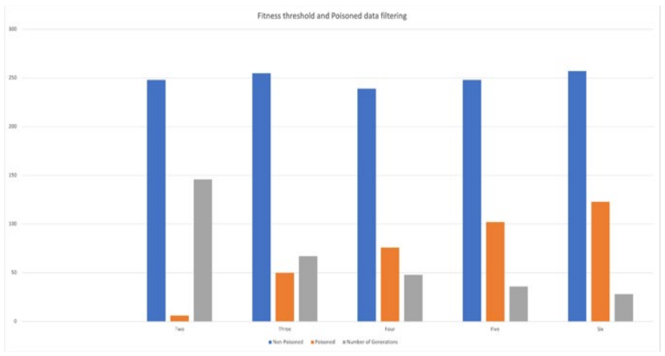

Figure. 5: Fitness (<4) vs Generation

5. Discussions, Limitations and Conclusions

5.1. The Search Space

As we discussed in previous sections Genetic Algorithms are efficient in finding solutions in the huge search space. How big the search space can be, let’s consider our use case. We have 800 sentences in our dataset. Out of 800 sentences, we will try to find the best set of sentences (candidates) that can produce the best results when paraphrased. Assume all the non-poisoned sentences (400 sentences in our dataset) can potentially produce better results and all the poisoned sentences can’t produce better results. Further, assume that we would like to select around 300 sentences out of 800. Then according to the combination formula 800 choose 300 (or 800C300) we are dealing with approximately **2.06 E+228** combinations. If we take a brute force approach, then we need to go through 2.06 E+228 combinations to find the best solution.

As we have already described in previous sections, the Genetic Algorithm doesn't go through all these combinations rather it tries to arrive at a better solution from the current solution. Therefore, genetic algorithms really excel when the search space is huge.

5.2. Future direction

The general idea here is to filter out certain contents or categorize content that can alter the behavior of the model. The same idea can be extended to build a content-aware model. In our work, we have demonstrated a way of filtering out copyrighted content by applying data poisoning techniques on data and applying Genetic Algorithms to reduce the poisoned content in the final data set.

For the demonstration purpose and due to resource constraints, we have used a simple model to generate text, perplexity score for the fitness computation, and many other parameters that were allowed within our resource limits. We can use more sophisticated models if resources are not a constraint, more complex functions can be used instead of just the perplexity score, different models can be employed for better pattern matching and different hyperparameters can be used to get better results or address different use cases. One of the interesting directions that can be explored is to find the correlation between the number of generations produced and the amount of poisoned content filtered. This correlation may provide some guidance in terms of the number of generations required when given the percentage of poisoned data that needs to be removed from the data set. This will also help in managing the computational cost.

5.3. Limitations

There are certain inherent limitations of Genetic algorithms [5] which our solution may suffer from are listed below-

1. It requires a special definition for various components of the algorithm. We need to devise appropriate representation for population, fitness function, Chromosome structure, as well as the selection crossover, and mutation operators tailored to the problem domain. This process can be demanding and time-intensive.
2. Genetic algorithms are influenced by a collection of hyperparameters, including population size and mutation rate, which govern their behavior. Determining the optimal values for these hyperparameters is not governed by strict rules when applying genetic algorithms to a specific problem.
3. Performing operations on populations, especially when dealing with large populations, and the iterative nature of genetic algorithms can be computationally demanding and time-consuming before achieving a satisfactory outcome. However, there are strategies to alleviate these challenges. Making appropriate choices for hyperparameters, such as population size and mutation rate, can optimize the algorithm's performance. Additionally, implementing parallel processing techniques can distribute the computational workload across multiple processors or machines, speeding up the execution. Furthermore, in certain cases, caching intermediate results can be beneficial, allowing the reuse of previously calculated values and reducing redundant computations. In our implementation, we computed certain results and cached them to avoid computing them every time we needed them.
4. When a single individual in the population significantly outperforms the rest in terms of fitness, there is a risk that it will dominate the entire population, resulting in premature convergence to a local maximum rather than exploring the global solution space. To avoid this issue, it is crucial to preserve the diversity within the population.

5. The application of genetic algorithms does not provide a guarantee of finding the global maximum for the given problem. However, this holds true for most search and optimization algorithms unless there exists an analytical solution specifically tailored to the problem. In general, when genetic algorithms are properly applied, they are recognized for their ability to generate satisfactory solutions within a reasonable timeframe.

Other limitations are-

1. The amount of poisoned data left out in the final response depends on the threshold defined for the perplexity score. Therefore, the threshold is also a hyperparameter, and fine-tuning the threshold could be challenging.
2. All the contents that we would like to hide from the model need to be poisoned. Since there is no single source of content and virtually everyone holds copyright for some content it is hard to implement data poisoning techniques across content owners.
3. The current study focuses on protecting copyright contents and current implementation doesn't explore categorizing copyrights and detecting a specific copyright.

4. Note: we tried to capture limitations to the best of our knowledge, there can be other limitations as well which we haven't discovered yet.

5.4. Deployment

To apply the techniques outlined in this paper to real-world application, the content owner who holds the copyright must poison their content as described in the paper. As shown in the example, data poisoning is straightforward and doesn't require special expertise. A web interface can be provided to generate poisoned text, as demonstrated in [3].

Once contents are poisoned and if crawlers retrieve the content to feed it to the LLMs then the model wouldn't be able to learn the actual context instead will learn some garbled text. Therefore, the model wouldn't be able to generate a meaningful response from the learned text. For the model to find a better response when input consists of poisoned and non-poisoned data, we employed genetic algorithms. As described genetic algorithm requires a way of comparing the generated results. We achieved this using a perplexity score. We cached perplexity scores in memory but for the real-world application databases or caching solutions can be employed. One can use any suitable algorithm instead of a perplexity score.

5.5. Conclusions

When the threshold for the fitness function is high algorithm has to produce more generation. That is a more robust solution that needs more generation. Though Genetic algorithms are computationally extensive, they are far better than traditional algorithms when we would like to find a solution from the large search space.

Genetic algorithms may or may not provide the best solution but in most cases, the solution is good enough for the given problem domain. For example, in our case, we would like to filter out content that is not relevant. Even though some unwanted content is left out in the final solution the ML model should be able to tolerate that noise.

References

[1] Josh A Goldstein, Girish Sastry, Micah Musser, Renee DiResta, Matthew Gentzel, and Katerina Sedova. Generative language models and automated influence operations: Emerging threats and potential mitigations. arXiv preprint arXiv:2301.04246, 2023.

[2] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. Advances in neural information processing systems, 30, 2017.

[3] Nicholas Boucher, Ilia Shumailov, Ross Anderson, and Nicolas Papernot. Bad characters: Imperceptible nlp attacks. In 2022 IEEE Symposium on Security and Privacy (SP), pages 1987–2004. IEEE, 2022.

[4] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. arXiv preprint arXiv:1910.13461, 2019.

[5] Wirsansky, E., 2020. Hands-on genetic algorithms with Python: applying genetic algorithms to solve real-world deep learning and artificial intelligence problems. Packt Publishing Ltd.

[6] Chen, W., Ramos, K., Mullaguri, K.N. and Wu, A.S., 2021. Genetic al gorithms for extractive summarization. arXiv preprint arXiv:2105.02365.

[7] Manzoni, L., Jakobovic, D., Mariot, L., Picek, S. and Castelli, M., 2020, June. Towards an evolutionary-based approach for natural language processing. In Proceedings of the 2020 Genetic and Evolutionary Computation Conference (pp. 985-993).

[8] Wallace, E., Zhao, T.Z., Feng, S. and Singh, S., 2020. Concealed data poisoning attacks on nlp models. arXiv preprint arXiv:2010.12563.

[9] Xiang, T., Xie, C., Guo, S., Li, J. and Zhang, T., 2021. Protecting Your NLG Models with Semantic and Robust Watermarks. arXiv preprint arXiv:2112.05428.

[10] Pajola, L. and Conti, M., 2021, September. Fall of Giants: How popular text-based MLaaS fall against a simple evasion attack. In 2021 IEEE European Symposium on Security and Privacy (EuroS&P) (pp. 198-211). IEEE.

[11] Michel, P., Li, X., Neubig, G. and Pino, J.M., 2019. On evaluation of adversarial perturbations for sequence-to-sequence models. arXiv preprint arXiv:1903.06620.

[12] Russo, A., 2023, June. Analysis and Detectability of Offline Data Poisoning Attacks on Linear Dynamical Systems. In Learning for Dynamics and Control Conference (pp. 1086-1098). PMLR.

[13] Evans, O., Cotton-Barratt, O., Finnveden, L., Bales, A., Balwit, A., Wills, P., Righetti, L. and Saunders, W., 2021. Truthful AI: Developing and governing AI that does not lie. arXiv preprint arXiv:2110.06674.

[14] Bang, Y., Cahyawijaya, S., Lee, N., Dai, W., Su, D., Wilie, B., Lovenia, H., Ji, Z., Yu, T., Chung, W. and Do, Q.V., 2023. A multitask, multilingual, multimodal evaluation of chatgpt on reasoning, hallucination, and interactivity. arXiv preprint arXiv:2302.04023.

[15] Megahed, F.M., Chen, Y.J., Ferris, J.A., Knoth, S. and Jones-Farmer, L.A., 2023. How generative ai models such as chatgpt can be (mis) used in spc practice, education, and research? an exploratory study. Quality Engineering, pp.1-29.

[16] Borji, A., 2023. A categorical archive of chatgpt failures. arXiv preprint arXiv:2302.03494.

[17] Sheera Frenkel, "Iranian Disinformation Effort Went Small to Stay Under Big Tech's Radar," *New York Times*, June 30, 2021,

https://www.nytimes.com/2021/06/30/technology/disinformation-message-apps.html.

[18] Xiang, Tao, Chunlong Xie, Shangwei Guo, Jiwei Li, and Tianwei Zhang. "Protecting Your NLG Models with Semantic and Robust Watermarks." *arxiv:2112.05428* [*cs.MM*], December 10, 2021. https: //doi.org/10.48550/arxiv.2112.05428.

[19] Sablayrolles, Alexandre, Matthijs Douze, Cordelia Schmid, and Hervé Jégou. "Radioactive data: tracing through training." *37th International Conference on Machine Learning, ICML 2020* PartF168147-11 (February 3, 2020): 8296–8305. https://doi.org/10.48550/arxiv.2002.00937.

[20] Ziegler, Z.M., Deng, Y. and Rush, A.M., 2019. Neural linguistic steganography. *arXiv preprint arXiv:1909.01496*.

[21] Lewis, M., Liu, Y., Goyal, N., Ghazvininejad, M., Mohamed, A., Levy, O., Stoyanov, V. and Zettlemoyer, L., 2019. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *arXiv preprint arXiv:1910.13461*.