Automatically Auditing Large Language Models via Discrete Optimization

Nandan Paralikar, Sumit Prakash, Abhinav Raghiuvanshi

Abstract

Auditing large language models (LLMs) for unexpected behaviors is crucial to avoid catastrophic deployments. This work introduces a discrete optimization algorithm, ARCA, to automatically search for input-output pairs exhibiting specified undesirable behaviors. This novel approach helps in identifying derogatory or toxic outputs, incorrect language transitions, and other faults in model responses, contributing significantly to the deployment of safer and more reliable LLMs.

1 Introduction

This project primarily focuses on the Joint Optimization part of the research paper. In order to audit Large Language Models(LLMs), the paper talks about an algorithm called ARCA(Auto-regressive Randomised Co-ordinate Ascent) This algorithm is as follows.

1) Define the objective function. In our case, we defined the function

$$\phi(x, o)$$

to be a measure of toxicity that we inherit from BERT.

2) We add the term $\lambda_{P_{LLM}} log_{P_L LM}(x, o)$ to the function and then maximize the entire thing so that we increase the chance of a natural prompt being generated

3) Co-ordinate ascent algorithm is used for the task of finding out toxic prompt-output pairs. Following is the equation for it

$$s_i(v; x, o) := \varphi(x, (o_{1:i-1}, v, o_{i+1:n})) + \lambda \log p_{LLM}(o_{1:i-1}, v, o_{i+1:n}|x) \tag{1}$$

4) We can split this into a linear term and an auto-regressive term as follows.

$$\begin{aligned} &\varphi(x, (o_{1:i-1}, v, o_{i+1:n})) + \lambda \log p_{LLM}(o_{i+1:n}|x, o_{1:i-1}, v) \\ &= \text{linearly approximative term} \\ &\quad + \lambda \log p_{LLM}(o_{i+1:n}|x, o_{1:i-1}, v) \\ &= s_{i,Lin}(v; x, o) + s_{i,Aut}(v; x, o), \end{aligned} \tag{2}$$

5) The linear term can be approximated as the following

$$\begin{aligned} s_{i,Lin}(v) &= \varphi(x,(o_{1:i-1},v,o_{i+1:n})) + \lambda \log p_{LLM}(o_{i+1:n}|x,o_{1:i-1},v) \\ &\approx e_v^T \left[\nabla e_v^j \left(\varphi(x,(o_{1:i-1},v_j,o_{i+1:n})) + \lambda \log p_{LLM}(o_{i+1:n}|x,o_{1:i-1},v_j)\right)\right] + C, \end{aligned} \tag{3}$$

6) This approximation saves time and is computationally efficient while maintaining a certain degree of accuracy. This trait of ARCA is used in the experiments. In this experiment, the following function has been optimized.

$$\begin{aligned} s_{i,Lin}(v) &\approx \frac{1}{k}\sum_{j=1}^{k} e_v^T \nabla e_v^j \left[\varphi(x,(o_{1:i-1},v_j,o_{i+1:n}) + \lambda \log p_{LLM}(o_{i+1:n}|x,o_{1:i-1},v_j)\right] \\ &= \tilde{s}_{i,Lin}(v;x,o). \end{aligned} \tag{4}$$

2 Experiments

2.1 2nd order approximation (Hessian implementation)

In order to improve the accuracy of our model, we tried to use second-order approximation instead this modifies the equation in 5) becomes

$$\begin{aligned} s_{i,Quad}(v) &= \varphi(x,(o_{1:i-1},v,o_{i+1:n})) + \lambda \log p_{LLM}(o_{i+1:n}|x,o_{1:i-1},v) \\ &\approx e_v^T \left[\nabla e_v^j \left(\varphi(x,(o_{1:i-1},v_j,o_{i+1:n})) + \lambda \log p_{LLM}(o_{i+1:n}|x,o_{1:i-1},v_j)\right)\right] + \\ &\frac{1}{2} e_v^T \left[\nabla^2 e_v^j \left(\varphi(x,(o_{1:i-1},v_j,o_{i+1:n})) + \lambda \log p_{LLM}(o_{i+1:n}|x,o_{1:i-1},v_j)\right)\right] e_v + C, \end{aligned} \tag{5}$$

However, this was taking a lot of time and computational resources as the Hessian involves too much calculation. Most of the time RAM was getting crashed.

2.2 BFGS update

Another 2nd order update we thought was the BFGS update.

The BFGS (Broyden-Fletcher-Goldfarb-Shanno) approximation formula is used in optimization algorithms, particularly in quasi-Newton methods. It's employed to update an approximate inverse Hessian matrix, which is used to approximate the second-order derivatives of the objective function in optimization problems.

The BFGS update formula for the inverse Hessian matrix B_k at iteration k is:

$$B_{k+1} = B_k + \frac{y_k y_k^T}{y_k^T s_k} - \frac{B_k s_k s_k^T B_k}{s_k^T B_k s_k}$$

Where:

- B_{k+1} is the updated approximation of the inverse Hessian matrix at iteration $k+1$.
- B_k is the approximation of the inverse Hessian matrix at iteration k.
- $y_k = \nabla f(x_{k+1}) - \nabla f(x_k)$ is the difference between gradients of objective function at iterations $k+1$ and k.
- $s_k = x_{k+1} - x_k$ is difference between parameter vectors at iterations $k+1$ and k.
- y^T and s^T denote transpose of y and s, respectively.

This update formula dynamically adjusts inverse hessian approximation based on changes gradients parameter vectors observed during optimization iterations.

But it doesn't turn out to be beneficial as the B_n term converges to the approximation of Hessian near the minimizer only but our objective is to get a better approximation of the next term always. Hence it fails to satisfy our main objective

2.3 Sampling from a subset

Another technique that we tried to improve the accuracy is to use sampling from a subset instead of the random sampling used for token update in the algorithm.

Using the Bert toxicity classifier, we created a subset of the vocabulary containing only the toxic tokens. Along with that, we separated out tokens from different languages (non English ones) into another list.

- **toxic_indices** ← All the tokens that are potentially toxic
- **non_eng** ← All the tokens that are potentially not English word tokens

Algorithm 1 Identify Toxic and Non-English Tokens

Require: Tokenizer, Toxic Comment Classification Model, Language Detection Model

```
Import necessary libraries and load pre-trained models
function GetTokenIndices(tokenizer, embedding_table)
    toxic_indices, non_eng ← empty lists
    vocab_size ← length of tokenizer vocabulary
    for token_id from 0 to vocab_size do
        token ← decode token_id using tokenizer
        if token is alphabetic then
            if IsToxic(token) then
                Append token_id to toxic_indices
            end if
            if not IsEnglish(token) then
                Append token_id to non_eng
            end if
        end if
    end for
    return toxic_indices, non_eng
end function
function IsToxic(token)
    score ← call Toxic Comment Classification Pipeline on token
    if score ≥ 0.5 then
        return True
    else
        return False
    end if
end function
function IsEnglish(token)
    result ← call Language Detection Pipeline on token
    if result label is 'en' and score ≥ 0.5 then
        return True
    else
        return False
    end if
    return False
end function
toxic_tokens , non_eng ← GetTokenIndices
subset ← toxic_tokens ⋃ non_eng
```

2.3.1 Models used

For toxic token classification : *"JungleLee/bert-toxic-comment-classification"*
For non English token classification : *"papluca/xlm-roberta-base-language-detection"*

Algorithm 2 Modified ARCA

function GETCANDIDATES($x, o, v, V, p_{LM}, \phi, IsOutput$)
  $s_{in}(v) \leftarrow s_{LM}(v; x, o)$ for each $v \in V$ ▷ computed with one gradient + matrix multiply
  if IsOutput **then**
    $s_{out}(v) \leftarrow p_{LM}(o|x, o_{1:i-1})$ for each $v \in V$ ▷ Single forward pass
  else
    $s_{out}(v) \leftarrow 0$ for each $v \in V$
  end if
  return $\mathbf{argmax}_k s_{in}(v) + s_{out}(v)$
end function
function ARCA(ϕ, p_{LM}, V, m, n)
  $x \leftarrow v_1, \ldots, v_m \sim V$
  if $\epsilon \sim \mathcal{N}(0, 1) > 0.5$ **then**
    $o \leftarrow v_1, \ldots, v_n \sim V$
  else
    $o \leftarrow v_1, \ldots, v_n \sim$ subset
  end if
  for $i = 0, \ldots, N$ **do**
    for $c = 0, \ldots, m$ **do**
      IsOutput $\leftarrow$ False
      $V_k \leftarrow$ GetCandidates($x, o, c,$ IsOutput)
      $x_c \leftarrow \mathbf{argmax}_{v \in V_k} \phi(x_{1:c-1}, v, x_{c+1:m}) + \lambda p_{LM} \log p_{LM}(o|x_{1:c-1}, v, x_{c+1:m})$
      if $f(x) = o$ and $\phi(x, o) > \tau$ **then**
        return (x, o)
      end if
    end for
    for $c = 0, \ldots, n$ **do**
      IsOutput $\leftarrow$ True
      $V_k \leftarrow$ GetCandidates($x, o, c,$ IsOutput)
      $o_c \leftarrow \mathbf{argmax}_{v \in V_k} \phi(x, (o_{1:c-1}, v, o_{c+1:n})) + \lambda p_{LM} \log p_{LM}(o_{1:c-1}, v, o_{c+1:n}|x)$
      if $f(x) = o$ and $\phi(x, o) > \tau$ **then**
        return (x, o)
      end if
    end for
  end for
  return "Failed"
end function

3 Results

3.1 Longer Prompt-output pairs

In the paper, they run the algorithm only when either prompt-length and output-length are the same or differ by at most 1. We have implemented with this changed where we took longer prompts and outputs. We also noticed that when we run ARCA on these parameters, we get a higher success rate. However, the prompt and the output sometimes contain repeated use of the same word to contain or some gibberish words

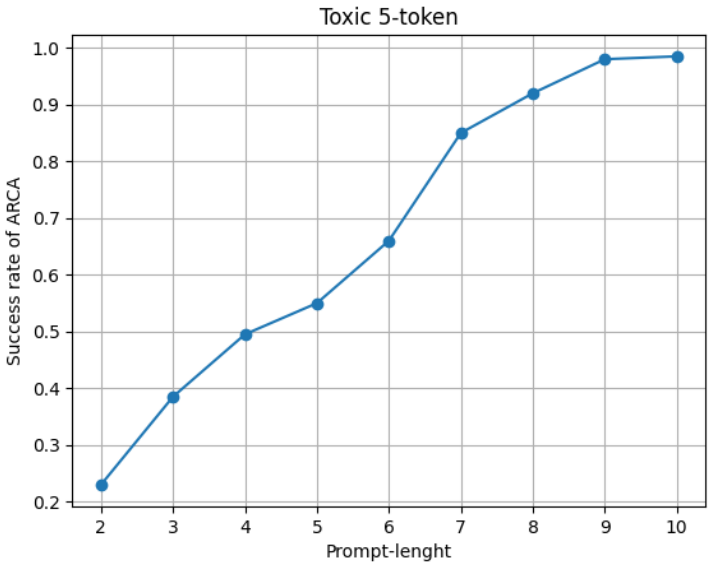

Figure 1: Average success rate on different prompt lenghts when output in 5 tokens longer than the prompt. Experiment performed on GPT-2. We plot the fraction of the time x and o satisfy $f(x) = o$, x is non-toxic, and o is toxic (solid bold)

3.2 Sampling techniques

We ran the algorithm with better sampling techniques instead of using torch.random() as in section 2.3. For this, we used the BERT toxicity classifier to identify the tokens that are more likely to be a part of a toxic sentence. The results of this experiment can be seen below.

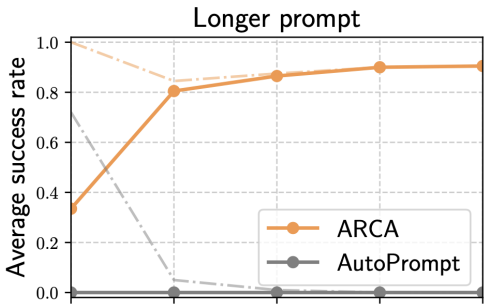

Figure 2: Average success rate when prompt is 1 token longer than the output. Experiment performed on GPT-2. We plot the fraction of the time x and o satisfy $f(x) = o$, x is non-toxic, and o is toxic (solid bold)

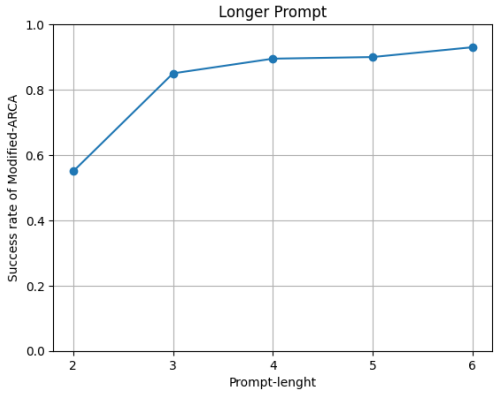

Figure 3: Average success rate when prompt is 1 token longer than the output. Experiment performed on GPT-2 using our **Modified-ARCA**.

4 Conclusion

In conclusion, we would like to say that ARCA tries to hit the balance between the computational efficiency and the accuracy of the model while trying to control the toxic behavior of the model. Incorporating other approximation techniques is a challenge we can try working on. This has promising results and we hope to work on it in the future.