Bearly Prompting:
Extracting Prompts by Inverting LLM Outputs

A Tensorflow reimplementation of output2prompt: a memory-efficient black-box approach for recovering user and system prompts from LLM outputs

Logan Rabe (lrabe), Rahul Mani (rmani1), Xinyi Wang (xwang435), Korgan Atillasoy (katillas)

Introduction

Our project focuses on reimplementing and extending the work presented in the paper *"Extracting Prompts by Inverting LLM Outputs"* by Zhang et al. (2024), which introduces **output2prompt**, a novel method for recovering the original prompts that generated large language model (LLM) outputs. Unlike prior approaches that depend on accessing hidden model internals such as logits or using unreliable jailbreak techniques, which are unreliable and hard to accomplish, output2prompt offers a black-box solution that requires only the output text to infer the underlying prompt. We chose this paper because it can help uncover the hidden instructions inside AI models—often proprietary system prompts—that govern LLM behavior, and provides a safer, more reliable way to uncover them. Our project reimplements the original PyTorch-based method in TensorFlow and also tests it on new examples to evaluate generalization.

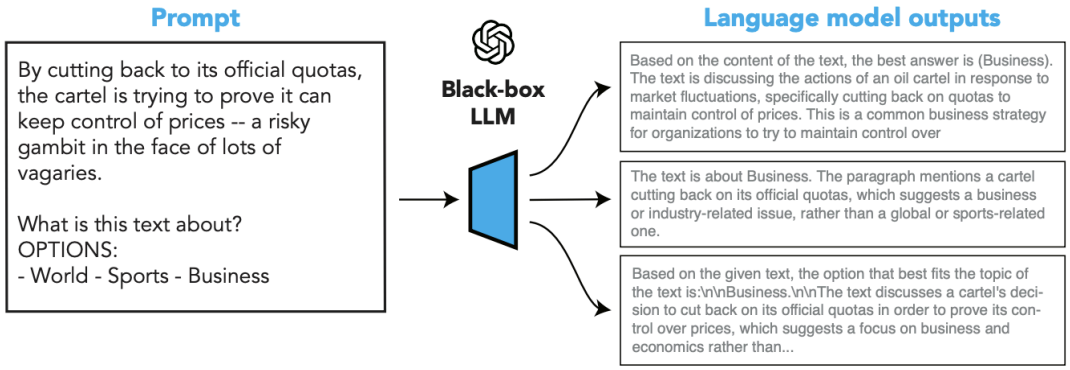
Methodology

We reimplemented the architecture detailed in the output2prompt paper, namely the two inversion models described: (1) a *user prompt* inversion model, which is trained to recover a user's full prompt from multiple LLM outputs, and (2) a *system prompt* inversion model, which is trained to recover the hidden system prompt given multiple LLM outputs generated via a known user prompt. Both models use the same T5-base encoder-decoder transformer architecture, with different sequence length configurations due to differences in the standard length of the target prompt type, i.e. user prompts are typically shorter and so the corresponding model uses a maximum sequence length of 64 tokens, as compared to the system prompt model's 256.

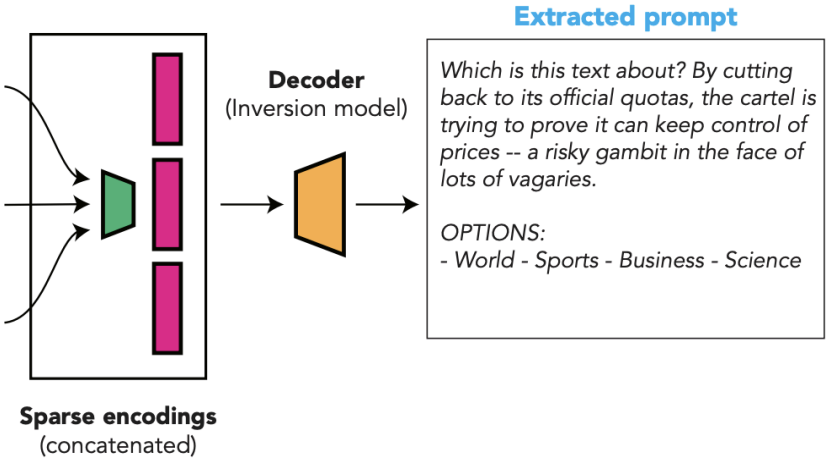
The core of the architecture is a transformer-based encoder-decoder model built on *T5-base*, which is pre-trained and subsequently fine-tuned. The T5-Base model is a pre-trained transformer from Google that treats every NLP task as a text-to-text problem and has about 220 million parameters. The transformer architecture processes words in an input sequence in parallel, allowing it to capture contextual relationships between all words in the sequence at once.

The model takes as input multiple LLM outputs and first encodes them using a *sparse attention encoder*. The sparse encoder processes each LLM output independently, wherein each

output only attends to itself. In other words, it only compares words within the same output to determine which words are most important in recovering the original prompt, rather than comparing all words across all LLM outputs. This design drastically reduces computational cost and memory usage. In fact, using full attention takes about four times as long to run while only improving cosine similarity by 0.3 (96.7 to 97). The encoder produces individual hidden representations for each output, also referred to as hidden features. These hidden features are mathematical representations of the model's internal understanding of what is most significant in determining the original prompt.



The resultant hidden features from each independently processed LLM output are then concatenated and passed to the decoder. The decoder generates the predicted prompt using *greedy autoregressive decoding*, i.e. it generates the prompt one token at a time, always picking the most likely next word at each step without exploring multiple options. To prevent longer than desired prompts, we constrain the maximum output sequence length of the decoder to 64 for user prompts and 256 for system prompts, consistent with the original paper.



In terms of training data, we trained both models on *approximately 85,000 prompt-response pairs* drawn from the *six datasets* used by the original paper. For user prompts,

we used Instructions-2M, ShareGPT, and Unnatural Instructions. For system prompts, we used Real GPTs, Awesome-ChatGPT-Prompts, and synthetic GPTs. The synthetic system prompts were generated by GPT-3.5 by providing it with the names and descriptions of real GPTs collected from the GPT Hunter database.

In the original paper, both inversion models were trained using PyTorch with bfloat16 precision on a single NVIDIA A40 GPU with 48GB of memory. The authors fine-tuned the T5-base model for 3 epochs using the Adam optimizer with a constant learning rate of 2e-4, applying teacher forcing during training and greedy decoding at inference time. For our reimplementation, we used *TensorFlow* instead of PyTorch. While we aimed to match the original setup as closely as possible in terms of optimizer choice, learning rate, and number of epochs, TensorFlow abstracts away hardware-specific configurations. Therefore, we did not manually specify the use of an A40 GPU or bfloat16 precision, but instead allowed TensorFlow to select optimal defaults based on the available environment. All other aspects of training, including the use of teacher forcing and evaluation using BLEU score and token-level F1 score remained consistent with the paper's methodology.

Results

Our results on training data (Instructions-2M and Synthetic GPTs) were close but slightly under the paper's results. We believe this is mainly due to not training on the full dataset. We generally outperformed the paper on the test datasets which could be due to the paper overfitting when training on the full dataset. As mentioned above, we were not able to report cosine similarity or LLM Eval without an OpenAI key, so it is difficult to know the true quality of our model. Full results can be seen below:

User dataset	type	CS	BLEU	Token F1	LLM Eval
Instructions-2M	original	96.7	56.9	79.5	82.9
ShareGPT	original	84.2	2.5	29.5	43.8
Unnatural Instructions	original	83.3	4.4	36.0	63.0
Instructions-2M	implemented		51.2	73.2	
ShareGPT	implemented		11.0	40.3	
Unnatural Instructions	implemented		13.6	45.5	

System dataset	type	CS	BLEU	Token F1	LLM Eval
Synthetic GPTs	original	98.1	36.8	73.8	99.0
Real GPTs	original	89.9	6.4	37.6	65.5
Awesome-ChatGPT-Prompts	original	83.9	2.1	28.8	77.2
Synthetic GPTs	implemented		36.2	70.6	
Real GPTs	implemented		0.8	37.9	
Awesome-ChatGPT-Prompts	implemented		3.8	29.1	

Challenges

In practice, we encountered some implementation challenges. The *accelerate* library used by the original codebase had changed since the paper's release, resulting in compatibility issues. We resolved these by downgrading the accelerate package to version 0.21.0. Additionally, the default batch size of 8 exceeded the memory limits of our local machines. To address this, we reduced the batch size to 1 and performed training and testing on a reduced subset of the dataset,

typically 1 percent, when running locally. In all local experiments, we trained for 1 epoch per model.

While the original paper evaluated model performance using multiple metrics, including BLEU, token-level F1, cosine similarity, exact match, and LLM-based scoring via the LLM Eval framework, our evaluation is limited to a subset of those metrics. In particular, we report *BLEU scores* and *token-level F1 scores*, both of which were used in the original paper and do not require external API calls. However, due to the lack of access to the necessary private API keys, we were unable to reproduce the cosine similarity and LLM Eval components of the evaluation. As a result, our reported results reflect only the metrics that could be computed locally, without reliance on proprietary model outputs or scoring services.

Reflection

We feel like our project turned out well as our biggest goal was simply a working model that was close to the paper. We didn't quite meet our base goal of 50 BLEU and 75 Token-F1 but we think that with more training this would be possible. Most of our other goals involved cosine similarity which we now know we cannot use.

Our model generally worked out as expected. We were able to keep the structure the same and our evaluation and training time were very similar to the paper. However, we underestimated the difficulty of implementing pytorch code in tensorflow. We thought this was the easier way to go since it reduced complexity, but there were also pytorch functions that we could not use in tensorflow. This is something we could have used more time to properly address.

Originally we planned to implement the code by each taking about ¼ of the files in the paper's repository. However, many of these files did not matter at all and we didn't need them to run the model. For example, some files addressed inversion from logits which is what this paper was improving upon with output2prompt, which doesn't use the model logits. If we could go back, we might collect which files are actually used by the model and divide them more evenly. We also would have given ourselves more time to merge the quarters we implemented which proved to be a tricky challenge.

If we had more time, we could have trained on the full dataset as well as implementing fine-tuning which the paper used to improve results. Additionally, we might be able to train and test on different data. For example, the paper samples 30K prompts and outputs from the Instructions-2M dataset. We only had access to the 30K sample, but with more time we could have taken our own sample of the dataset and tested for similar results.

One of the main things we learned from this project was how to interpret and run an academic paper. This paper specifically used a transformer model which was easy to understand

as we spent time on it in lecture. This was the first time many of us ran the code of an academic paper locally which proved to be not as difficult as we thought. We also learned a lot about the differences between tensorflow and pytorch which we saw when implementing in tensorflow. Finally, the poster presentations taught us a lot about how to communicate our results to a broad audience. It was also the first time doing computer science poster presentations for many of us but the outlines, reflections, and TA check-ins were good preparation for explaining our project concisely.