Problem Statement

Without a doubt, LLMs have skyrocketed in popularity. Their ability to generalize to many different kinds of tasks, from conversation, summarization, code generation, to now reasoning, as well as their accessible interfaces (natural language) make them one of the best outcomes from AI work ever. To make LLMs even more powerful, **agents** have been developed that equip LLMs with tools, giving them agency. Common agents are those with calculators, web search capability, and image generation. Agents use LLMs to "think," and then they use their tools to "act," which gives them feedback in text for them to "observe." Furthermore, multi-agent systems are being developed to automate more complicated tasks, as they allow for efficiency and specialization.

With new technology, as always, comes new security risks. We have already seen attacks on LLMs such as **prompt injection** and **jailbreaking**, and with their ability to act, LLM agents may open the door for further misuse. Multi-agent systems, with their added complexity, may add even more security holes. My problem statement is as follows:

> *What vulnerabilities are there within LLM agents and multi-agent systems? How may these new technologies be misused for harm? Furthermore, what action can be taken to mitigate and prevent such damages?*

Prompt Engineering

Settings

Temperature: how creative you want the LLM to be. Higher means selects more creative (lower chance) tokens, lower means less creative.

Top P: Tightens the token pool. p is the total percent you want to pool from. Lower value means less creative, higher value means more creative

Top K: Chooses from the top K tokens.

Note: It’s recommended you use temperature, top p, and top k disjointly.

Max length: control output of the model

Stop sequences: Tell the model to stop output when it outputs a specific phrase

Frequency penalty: Repeated tokens are discouraged (more tokens $\implies$ less consideration)

Presence penalty: Repeated tokens are discouraged; doesn’t change if there are more than 2 repeats.

Note: It’s recommended you use frequency penalty and presence penalty disjointly

Elements

Prompts can be broken up into 4 sections: **instruction, context, input, and output.** These are all self-explanatory.

Techniques

Zero-shot vs Few-shot

"shots" are how many examples you provide. These go in the context. Few shot improves the response of the model, even if the examples are wrong (think of it as improving the structure, which improves the output)

Chain of Thought (CoT)

Tell the model to "think step-by-step." Basically creates a chain of reasoning within the model, allowing it solve reasoning questions better. The context window is basically all the model has to work off of, so if it can give itself more context, that helps with reasoning.

Automatic Chain of Thought is when the LLM clusters questions that are similar, and when answer its CoT, samples from clusters. Reduces chance of choosing bad responses.

Meta Prompting

Force the model to follow a specific structure. Can be zero shot (just provide template) or few shot. Structure allows the model to give more preditable, desired answers.

Self-consistency (SC)

Ask the LLM to do CoT multiple times. Choose the answer that appears the majority of the time from the thought processes.

Generated knowledge

Before giving its response, ask the model to come up with its knowledge on the subject first. Allows the model to work with more context/knowledge. Can be zero shot or few shot

Prompt chaining

Break your prompt into multiple prompts. Feed the output of one as part of the input of the other

Tree of Thoughts

Tell the model to generate multiple reasoning paths, and continue with the ones that look most promising. Involves intermediate step in which model (can be same or different) evaluates responses as possible, likely, impossible, etc. Shown to be better than CoT and CoT with SC.

Retrieval Augmented Generation (RAG)

The model grabs more information (either from a database, search, etc) before making its response. Reduces hallucinations

Automatic Reasoning and Tool-Use (ART)

Puts tools into CoT. Follows examples from a *task library*.

Automatic Prompt Engineer (APE)

Ask a model to give a prompt that produces desired output for given input. Run those prompts through an LLM to see which ones generate the expected output with highest probability. Ask LLM to generate prompts with the same sentiment. Test again

Active-Prompt

Asks model to use CoT to solve questions from a dataset. Those with most uncertainty (eg most conflicting answers) are sent for human annotation to improve the model.

Directional Stimulus Prompting

Make a LM that generates hints for the LLM to generate better responses

Progam Aided Language Models (PAL Models)

Use a code interpreter with the model, and ask the LLM to generate & run code to solve problems where it can be solved with an algorithm/calculation

ReAct

A loop of thinking, acting (using a tool), observing the results of that action, then (optionally) repeating.

Reflexion

ReAct with another step where an LM (same or different) says whether the results are good or not to help the actor behave better.

Multimodal CoT Prompting

Tells the model to analyze each object, effectively putting them both in the context window, before asking the LLM what to do.

GraphPrompts

Prompting for graphs

Imprompter

Demo website

Synopsis

Imprompter is an attack method that involves attackers invoking tool-usage in LLM agents to reveal personally identifying information (PII), under the guise of an ambiguous prompt advertised with some other function (eg CV review). It uses gradient-based optimization techniques to make sure the syntax for tool invocation is correct, as well as make sure the prompt is hard to recognize. They define loss functions based on previous context, the prompt, and desired output. They also use perplexity to define the level of obfuscation. The prompt can also be a purposefully distorted image, which the authors note as naturally obfuscated in the image encoding. Results show ~80% effectiveness in revealing PII.

Vs Prompt Injection & Jailbreaking

Prompt Injection

Prompt injection is a technique in which attackers create prompts that users give to the LLM that lead to the attacker's desired behavior. Imprompter is similar to prompt injection, in that the attacker creates a prompt to be used by the user, but Imprompter obfuscates the prompt, arousing less suspicion.

Jailbreaking

Jailbreaking is when an attacker manages to get an LLM to break its own rules for safety & conduct. An advantage of jailbreaking is that people have found ways to jailbreak LLMs without knowing model weights. Imprompter, on the other hand, requires model weights (though the authors believe their work can be extended to closed-weight models), and doesn't force the model to do anything against the safety conduct; instead, the attacker tricks *the user* into revealing their own information.

Questions/notes

- This paper's focus (along with prompt injection) seem to be weaker to me than jailbreaking, as it requires users to put in the prompt themselves.
- In the paper, the LLM replies to the malicious prompt with an invisible 1x1 image. This, clearly, could raise suspicion, though the paper says users will likely note this is strange but move on. I bet it'll be less suspicious (and probably not so difficult) to have the prompt do as it's advertised as well. This increases its likelihood to be shared and will let people be less suspicious of it.
- For the same reasons the paper states, adversial images seem to better at not raising suspicion compared to prompts, as they don't reveal anything as to what's going on behind the scenes (in the prompt) at all. You could even distort an image that looks like it's supposed to do the right thing (eg an image with text telling the LLM what to do).
- What would be the best way to prevent this? The paper proposes some ideas, but none of them seem very good (ie they're all restrictive). Maybe there's a way to encrypt/hide tool invocation tokens (in code and in output), making it so users cannot force the LLM to make those invocations.

Related Work/future reading list

- greedy coordinate gradient

Grant Proposal Management System

Purpose

Writing grants has a lot of moving parts: a Primary Investigator (PI) needs to decide Co-PIs, Senior Personnel (SP), etc. and then once they submit their proposal in a Proposal Datasheet (PDS), several people (Department heads, chairs, review boards, etc.) must approve/disapprove the proposal. This process involves many users, some of whom have different access abilities than others (for example, the PI can create/delete the PDS, but Co-PIs should not), and it also has many moving parts/steps, some of which can be done in parallel. When scaled up to a college/university level, where multiple faculty from several departments are sending proposals through, it can be difficult to keep track of who's allowed to do what to which document, as well as which stage each document is in. Grant Proposal Management Systems aim to streamline & automate this system.

This paper focuses on making secure GPMS

More specifically, it wants to make sure that people have the right capabilites at all times. Access is not static: for example, a PI is able to create/delete a PDS before they submit it for approval, but once they submit it for approval, they cannot create/delete it until after the proper parties review it. Hence, maintaining proper access is not an easy task.

Forms of access control

Role Based Access Control (RBAC)

This is the classical way to assign access control. Roles have specific permissions, and each user is assigned to a role. Who gets to do what is clearly outlined by their role, but as systems get more complicated, the number of roles will increase, making the system harder to understand.

Attribute Based Access Control (ABAC)

ABAC works the following way: each user is given some attributes (job title, date started, etc.) and each attribute has a set of permissions associated with it. A specific user's permission to an object is then decided by their attributes in a conservative way. This allows for much more granularity than RBAC–a role can just be a combination of attributes–but also makes it much more difficult to immediately see who is able to do what (you'd need to comb through their attributes)

Next Generation Access Control (NGAC)

NGAC aims to bridge the gap between RBAC and ABAC. Users and objects have attributes still, but attributes are also related to other attributes, allowing for us to create graphs of access, where nodes are users, user attributes, objects, and object attributes (as well as policies that define who can do what), and edges are relations between users, relations between objects, and permissions user attributes are given for specific objects. With this, you still assign attributes to users, allowing for that same level of granularity as ABAC, and to find what permissions that user has, you can just do a walk along the graph you've made, allowing for that fast recognition of permission that RBAC gives.

This paper focuses on making a GPMS that uses NGAC for its clear advantages in a complex network of users and objects.

Obligations

Obligations are events that are associated with user actions. For example, when a PI assigns a Co-PI, they want that Co-PI to also have read/write permission for their PDS (the PI is "obligated" to give access), so an event is triggered whenever a PI assigns a Co-PI to automatically give that Co-PI read/write permission. This allows for the dynamic access control that is required in GPMSs.

This paper's goals

This paper is focused on making a fleshed out GPMS with NGAC and obligations. It provides the necessary specification and introduces a RESTful API, along with a website, the authors have created that implements this specification. In its specification, the paper describes the policies, actions, users, and general flow that is necessary for a proper GPMS, and formalizes all of this in set notation.

My Questions

- This paper seems to be more of an application, or a proof of concept, of NGAC + Obligations, which it claims has never been done before. This may be true, but the idea doesn't seem too novel to me: Obligations (at least from my understanding) are meant to be used with access control, so NGAC + Obligations is not too farfetched. Certainly the scale of this project is impressive, but it's not clear to me: what's the new idea that's being introduced here?
- The authors mention they have thoroughly tested their application with JUnit and Selenium. I'd like to see these tests, especially to understand what are common bugs that were planned for as well as edge cases. I think from a cybersecurity standpoint, this is very important.
- I am not sure how to apply agents to this. It seems that this is already a nearly fully autonomous framework, and since everything is rule based, it's reliable and doesn't make sense for it to be replaced by agents. Maybe we could use agents to create the rules, or simulate/test it?

Forward Passes and Back Propagation

This is the essential process of deep learning.

First, the Chain Rule

We learned in Calc I that for one variable $x$,

$$\frac{d}{dx}f(g(x)) = \frac{df}{dg} \cdot \frac{dg}{dx}$$

or alternatively,

$$\frac{d}{dx}f(g(x)) = f'(g(x))g'(x)$$

With multiple variables, consider vectors

$$\mathbf{u} \in \mathbb{R}^{\gtrdot}, \mathbf{v} \in \mathbb{R}^{\ltimes}$$

and functions (or in the case of DL, linear transformations)

$$g : \mathbb{R}^{\gtrdot} \to \mathbb{R}^{\ltimes}, f : \mathbb{R}^{\ltimes} \to \mathbb{R}, f \circ g : \mathbb{R}^{\gtrdot} \to \mathbb{R}$$

So that $g(\mathbf{u}) = \mathbf{v}, f(g(\mathbf{u})) = f(\mathbf{v})$. If we want to find the partial derivative $\frac{\partial f}{\partial u_i}$, that would be

$$\frac{\partial f}{\partial u_i} = \frac{\partial f}{\partial v_1} \cdot \frac{\partial v_1}{\partial u_i} + \frac{\partial f}{\partial v_2} \cdot \frac{\partial v_2}{\partial u_i} + \ldots + \frac{\partial f}{\partial v_n} \cdot \frac{\partial v_n}{\partial u_i}$$

A way to think about this is to say: OK, $f$ is affected by $\mathbf{v}$, which is affected by $\mathbf{u}$. If I want to see how $u_i$ affects $f$, then I should see how $\mathbf{v}$ affects $f$ first. We can do this component-wise. Then for each component, let's see how $u_i$ affects that component, $v_j$. Multiplying the two gives us how $u_i$ affects $f$ for that specific component $v_j$. Since $\mathbf{v} = \mathbf{v_1} + \mathbf{v_2} + \ldots + \mathbf{v_n}$, where $\mathbf{v_i} \in \mathbb{R^n}$ filled with $0$ in all but the $i^\text{th}$ spot, where it's $v_i$, it would make sense to add up those partial derivatives. Note that this is not a rigorous proof, just some intuition as to where this comes from.

It follows from here that the gradient is

$$\nabla f = \begin{bmatrix} \frac{\partial f}{\partial u_1} \\ \frac{\partial f}{\partial u_2} \\ \vdots \\ \frac{\partial f}{\partial u_m} \end{bmatrix}$$

Now, Deep Learning

Basically, the way a neural network is a model that tries to solve a problem based on data. We formulate problems this way: given an input, you expect an output. We'll take our data and format it into vectors: the input is an $\mathbf{u} \in \mathbb{R^m}$ vector and the output is an $\mathbf{v} \in \mathbb{R^n}$ vector. Maybe there's a linear relationship between the input and output! So we try to find $A \in \mathbb{R^n} \times \mathbb{R^m}, \mathbf{b} \in \mathbb{R^n}$ such that

$A\mathbf{u} + \mathbf{b} \approx \mathbf{v}$ *Note:* A and $\mathbf{b}$ are not given to us. We are only given $\mathbf{u}$, and we have to **find** the A and $\mathbf{b}$ that get us close to $\mathbf{v}$. How to do this is described in the back propagation section.

But as we see in real life, not all relationships are linear! So we need add an activation function (ReLU, sigmoid) that is non-linear. Let's use the sigmoid function. So now we have:

$$\mathbf{z} = \sigma(A\mathbf{u} + \mathbf{b})$$

Maybe this is good if we want our output components to be between 0 and 1 (as that's the range for the sigmoid function), but that's not always the case. Well, we have this new vector $\mathbf{z}$, so let's put that through a linear transformation to hopefully fit $\mathbf{v}$!

$$A'\mathbf{z} + \mathbf{b}' \approx \mathbf{v}$$

Note: Although this is a linear transformation, we're not gonna get a line, since we had that non-linear activation function to get $\mathbf{z}$! So the hope is that this fits $\mathbf{v}$ even better!

Forward pass

And the people who came up with neural networks were pretty clever in that they applied this logic multiple times. I'm gonna be pretty lazy with my notation here, so feel free to follow somewhere else if you don't understand, but the idea is to have multiple linear transformations & additions of however many dimensions you want, passing them through activation functions along the way, as long as the sequence is able to take in some $\mathbf{u}$ and give out some $\hat{\mathbf{v}}$, which looks like this:

$$\mathbf{z_1} = \sigma(A_1\mathbf{u} + \mathbf{b_1}), \quad \mathbf{z}_i = \sigma(A_i\mathbf{z}_{i-1} + \mathbf{b_i}), \quad A_l\mathbf{z}_l + \mathbf{b}_l = \hat{\mathbf{v}} \approx \mathbf{v}$$

where l is 1 more than the number of layers there are. Going through this is called a **forward pass**.

The cost function

Ok, so for some input, we have a guess at what the output could be and the actual output. So it makes sense to want to know how close we were. Well vectors are really just points in a space, so we could find the distance between them. There are a lot of ways to compute distance, but one of them would be to subtract terms and then square the difference. From here, we'll just take the average of the squared differences: this is called **mean squared error (MSE)**. If we have d data points, the MSE would be:

$$\frac{1}{d}\sum_{i=1}^{d}(\hat{\mathbf{v}}_i - \mathbf{v}_i)^T(\hat{\mathbf{v}}_i - \mathbf{v}_i)$$

We could also (maybe more obviously) use absolute value (or any other norm), but squaring distances has the nice effect of emphasizing points that are farther away.

So now we have this nice expression for our **cost**. We'd like to minimize the cost. How do we do this? Well, first it might help to make this a function. Most immediately, we can make our cost function c take in guesses and actual values:

$$c(\hat{\mathbf{v}}, \mathbf{v}) = \frac{1}{d}\sum_{i=1}^{d}(\hat{\mathbf{v}}_i - \mathbf{v}_i)^T(\hat{\mathbf{v}}_i - \mathbf{v}_i)$$

where $\hat{\mathbf{v}}$ is a collection of d guesses and $\mathbf{v}$ is a collection of d corresponding actual values.

We're given the actual values, but we can't actually control the guesses, at least not directly. What we *can* control is how we get those guesses. And how did we get those guesses? With $\mathbf{A} = A_1, A_2, \ldots, A_l$ and $\mathbf{b} = \mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_l$. So really, our cost function should take in the $\mathbf{u}, \mathbf{A}, \mathbf{b},$ and $\mathbf{v}$, and give us back the MSE. Moreover, we can only change $\mathbf{A}$ and $\mathbf{b}$, so let's set $\mathbf{u}$ and $\mathbf{v}$ to be constant in c:

$$c(\mathbf{A}, \mathbf{b}) = \frac{1}{d}\sum_{i=1}^{d}(\hat{\mathbf{v}}_i - \mathbf{v}_i)^T(\hat{\mathbf{v}}_i - \mathbf{v}_i)$$

where $\mathbf{u}, \mathbf{v}$ are fixed and $\hat{\mathbf{v}}$ is calculated using $\mathbf{u}, \mathbf{A}, \mathbf{b}$.

Gradient descent

Now we have a cost function that takes in two variables (or many variables if you think of each component of $\mathbf{A}$ and $\mathbf{b}$ to be a variable) and turns it into one. We'd like to minimize this: to do this, we're going to use **gradient descent**.

I mentioned the gradient in the chain rule section. Basically, it's the partial derivative of a function for each variable in that function. My claim is that the gradient tells us (locally) the way to move so that we travel the farthest, *and* how much how to move by. In other words, the gradient shows the direction (and magnitude) of *steepest ascent*.

I think it's easier to first see how the gradient shows magnitude. If you think about taking a step in each dimension along the current trajectory and seeing where that takes you from your function f, (aka the partial derivative), then you end up with the gradient. If a step makes you go higher, then the partial derivative will go higher, and vice versa. So it makes sense that larger values in the gradient correspond to bigger changes in your function.

Now to see why the gradient shows the direction of steepest of ascent, we'll consider all unit vectors at our point. We know the gradient, which can be thought of as taking a step in all directions and seeing where it takes us. So which unit vector will give us our steepest ascent? One way to think about how far will go is by projecting that unit vector onto the gradient. For example, let's consider $\nabla f = \begin{bmatrix}1 \ 0\end{bmatrix}$ and and the unit vectors $\begin{bmatrix} 1 \ 0 \end{bmatrix}, \begin{bmatrix} 0 \ 1 \end{bmatrix}$. Well the first vector is taking a step in the x dimension, so f is expected to increase by 1. The second vector takes a step in the y dimension, which the gradient shows as having no change, we f is expected to not change. So basically, we take the value in each component of our unit vector and multiply it by the corresponding component in the gradient: this is just the dot product! And so now we can answer our question, which unit vector, when dotted with the gradient, will give us the greatest value? And if we think of the dot product in terms of projections, we can ask the related question: which unit vector, when projected onto the gradient vector, will have the largest magnitude? And the answer to this is the gradient normalized. So therefore the direction of steepest ascent is the gradient itself!

And if the gradient shows us the direction of steepest ascent, then its negative should be the direction of *steepest descent*.

So why do we care? What are we going down? Well, if you remember, we want to minimize our cost function. One way to do this, if you've taken multi-variable calculus, would be to find where the gradient is 0, and consider each of those points (you can do additional tests to knock out some points), as these will all be local minima. But as functions get more and more complicated, finding the local minima will more annoying, and local minima aren't always global minima.

So instead, we're going to take the route of gradient descent. We'll start out at a random point in our space $\mathbf{x}_0$. We take the negative of the gradient there, and we'll take a step in that direction:

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \nabla f(\mathbf{x}_n)$$

And often, we want to control our granularity, so we introduce a parameter called the *learning rate*, that just adjusts the size of the steps we take.

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \gamma \nabla f(\mathbf{x}_n)$$

And we continue this process, either for a fixed number of steps or when the gradient gets small enough.

I must note that while this is faster than the method of finding local minima (and always possible given that our function is differentiable) and may find global minima, we have the problem of getting "stuck" in a local minimum. I mean, if we're at a local min, then the gradient will be 0 and we won't move, even if we're a a "high point" relative to the rest of the graph.

Back propagation

So now we have our method of minimizing, we just need to minimize our cost function. Well how do we do that? We have to first find our gradient, which will be populated by

$\dfrac{\partial c}{\partial A_i} \quad \forall A_i \in \mathbf{A}$ and

$$\frac{\partial c}{\partial \mathbf{b}_i} \quad \forall \mathbf{b}_i \in \mathbf{b}$$

But taking these partials won't be so easy! I mean, some $A_i, , \mathbf{b}_i$ could be used way earlier, meaning we've applied so many functions (linear transformations and shifts, sigmoid functions) to get from there to our final result in c. So that's where the chain rule comes in! We just need to **chain (or propagate) backwards** until we get the partial we want!

So now we have the gradient, we can finally do gradient descent. we get $\mathbf{A}, , \mathbf{b}$ that minimize c, so we’ve fitted our data to the expected output. Hooray!