

論文要約

LLM関連

概要: RAGの評価方法の提案

タスクに関連する文書コーパスに基づく多肢選択式問題から自動生成された合成試験にRAGをスコアリングすることで実施。試験の品質とタスク特化型精度に関する情報量を推定するために項目反応理論 (IRT) を活用することで試験を段階的に改善します

技術や手法

本研究で説明されている技術や手法は次の通りです:

1. **試験生成**:

- LLMを用いて、文書コーパスに基づく多肢選択式試験問題を自動生成。
- 各質問には1つの正解と複数の選択肢が含まれます。
- 各文書から質問候補を生成し、NLPベースのフィルターを適用して低品質な質問を除外。
- 試験生成プロンプト例

```markdown

Human: Here is some documentation from {task\_domain}: {documentation}. From this, generate a difficult multi-form question for an exam. It should have 4 candidates, 1 correct answer and explanations.

Syntax should be:

Question: {question}

A) {candidate A}

B) {candidate B}

概要: LLMでCoTを使用すると説明能力を向上させる半面、出力が長くなり応答に時間がかかるため、出力長を制御するプロンプトエンジニアリング制約付きプロトを与え、出力の簡潔さと応答時間の予測可能性を向上させます。

CoT (CCoT)を紹介。出力の長さを制約するプロンプト

つくり方は、Let's think a bit step by step の後にlimit the answer length to 45 words. のような制限の指定をするだけ

### 技術や手法

1. \*\*新しい評価指標の提案\*\*

- \*\*硬直な簡潔精度 ( Hard-k Concise Accuracy: HCA)\*\*: 指定された長さ k以下の正確な出力の割合を測定します。
- \*\*柔軟な簡潔精度 ( Soft-k Concise Accuracy: SCA)\*\*: 長さkを超える正確な出力に対して減衰因子  $\alpha$ を用いてペナルティを課します。
- \*\*一貫した簡潔精度 ( Consistent Concise Accuracy: CCA)\*\*: 出力長のばらつき  $\sigma$ に基づいて SCAをさらに調整します。

2. \*\*制約付き CoT (CCoT) の導入\*\*

- \*\*CCoTプロンプト\*\*: LLMに対して出力の長さを制約するプロンプトを与え、出力の簡潔さと応答時間の予測可能性を向上させます。

概要: RAGと長文コンテキスト( LC)の比較を行いました。結果として、リソースが十分にあれば LCが平均的な性能で RAGを上回ること、RAGは大幅に低コストであるという利点があること、この結果を基にモデルの自己反省に基づいてクエリを RAGまたはLCにルーティングする SELF-ROUTEという方法を提案。

計算コストを大幅に削減しながら、 LCと同等の性能を維持できます。

### 技術や手法

以下の3つの手法の比較を行っています

1. \*\*RAG (Retrieval Augmented Generation)\*\*:

- クエリに基づいて関連情報を検索し、 LLMがその情報を使用して応答を生成する。
- クエリに関連する情報を取得し、 LLMの注意を必要なセグメントに集中させることで、無関係な情報による注意の分散を防ぐ。
- 計算コストが低い。

2. \*\*長文コンテキスト( LC) LLMs\*\*:

- 大規模な事前学習により、長文コンテキストを直接理解する能力を持つ。
- 例: Gemini-1.5(最大1百万トークンを処理可能)、 GPT-4(128kトークンを処理可能)。

3. \*\*SELF-ROUTE\*\*:

- クエリを RAGまたはLCにルーティングする方法。
- モデルの自己反省に基づいてクエリが回答可能かどうかを予測し、回答可能な場合は RAGを使用し、そうでない場合は LCを使用する。

概要: LLMのエージェント tulip agentを紹介

ツールの説明をシステムプロンプトにエンコードせず、また全体のプロンプトを埋め込むことなく、再帰的にツールライブラリから適切なツールを検索することで実現します

### 技術や手法の詳細説明

### タスク分解

**概要**:

ユーザーからの自然言語クエリを受け取り、 LLM(大規模言語モデル)が高レベルのタスクを理解し、細分化されたサブタスクに分解します。これにより、具体的かつ実行可能な単位でタスクを処理できます。

**詳細**:

1. **ユーザークエリの受信** :

- エージェントはユーザーからの自然言語による質問や要求を受け取ります。
- 例: "What is 45342 multiplied by 23487 plus 32478?"

2. **タスク分解モデル (Mtd)**:

- このモデルは、受信したクエリを解析し、それをサブタスクに分解します。

概要: LLMを1つまたは複数の KGで拡張できるファインチューニングなしでゼロショットな Tree-of-Traversalsを紹介 KGを利用して可能な思考とアクションをツリー検索で行うことにより、高信頼の推論経路を見つけます

### 1. Tree-of-Traversals アルゴリズム

Tree-of-Traversalsは、ブラックボックス LLMを複数のナレッジグラフ( KG)で拡張するゼロショット推論アルゴリズムです。このアルゴリズムは、 KGとインターフェースをとるアクションを LLMIに装備し、ツリー検索を行うことで高信頼の推論経路を見つけ出します。

### 1.1 知識グラフインターフェース (Knowledge Graph Interface)

Knowledge Graph Interfaceは、LLMがKGと相互作用するためのメソッドを提供します。

- `initialize(q) → E0`: クエリ qを入力として受け取り、qからエンティティを抽出し、KGに存在するリンクされたエンティティ E0を返します。
- `get\_relations(Eselected) → Roptions`: 選択されたエンティティの集合 Eselectedを受け取り、KG内の利用可能なリレーションタイプ Roptionsを返します。
- `get\_edges(Eselected, r) → Tadded, Eadded`: 選択されたエンティティの集合と選択されたリレーションタイプ rを受け取り、そのリレーションタイプを持つエッジを返します。また、新たに追加されたエンティティ Eaddedも返します。

### 1.2 アクションステートマシン (Action State Machine)

# ThinkRepair: Self-Directed Automated Program Repair ThinkRepair: 自律型自動プログラム修復

概要: 自動プログラム修復 (APR) アプローチとして LLM を使用する ThinkRepair を提案。2 フェーズで構成され収集フェーズでは CoT プロンプトを使い事前に固定された知識を自動収集、修正フェーズでは、few-shot のための例を提示しテスト情報のフィードバックを追加することでバグ修正を行います

## ThinkRepair アルゴリズムと手法

### 収集フェーズ

このフェーズは、様々な考え方のチェーン (Chain-of-Thought, CoT) を収集し、知識プールを構築することを目的としています。

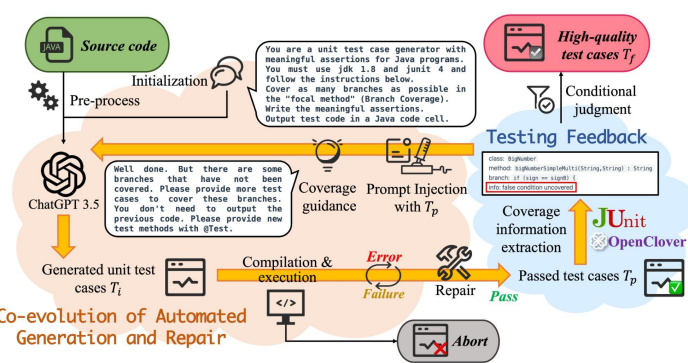
1. \*\*プロンプトの準備\*\*

- \*\*英語\*\*: The prompt used in ThinkRepair involves four important components:
  1. Role Designation: ThinkRepair starts a role for LLM with an instruction like “You are an Automated Program Repair tool”.
  2. Task Description: LLM is provided with the description constructed as “// Provide a fix for the buggy function”. Since we illustrate an example in Java, we use the Java comment format of “//” as a prefix.
  3. Buggy Function: ThinkRepair provides the buggy function to LLM in our single-function fixing scenario. We also prefix the buggy function with “// Buggy Function” to directly indicate LLM about the context of the function.
  4. Chain-of-Thought Indicator: LLM is instructed to think step-by-step when fixing a bug. In this paper, we follow the best practice in previous work and adopt the same prompt named “Let’s think step by step”.

- \*\*日本語\*\*: ThinkRepair で使用されるプロンプトには、次の四つの重要なコンポーネントが含まれます。

1. 役割の指定: ThinkRepair は「あなたは自動プログラム修復ツールです」という指示で、LLM に役割を与えます。

概要: 単体テストを LLMを使用して生成する TestARTを提案。テンプレートベースの事前に定義された修復パターンを使用してコードのバグを自動的に修正。修正されたコードや追加の指示をプロンプトインジェクションとして LLMIに与え、次の自動生成ステップを誘導します。テストケースからカバレッジ情報を抽出し、テストフィードバックとして使用します。



### 1. 前処理

\*\*目的\*\*：ソースコードを LLMが処理しやすい形にする。

- \*\*コメント削除\*\*：コード内のコメントや余分な空白行を削除します。これは、コードの冗長性を減らし、LLMが必要な情報に集中できるようにするためです。

- \*\*コード圧縮\*\*：メソッド署名のみを残し、焦点となるメソッドの完全なメソッドボディを保持します。これにより、LLMが扱うテキストの長さを適切に保ち、無関係な情報の干渉を減らします。

### 2. 生成と修復の共進化

```
Focal Method--getShortClassName
public static String getShortClassName(String className) {
 if (!className.startsWith("(")) {
 while (className.charAt(0) == '(') {
 className = className.substring(1);
 arrayPrefix.append("(");
 }
 if (className.charAt(0) == 'L' && className.charAt(className.length() - 1) == ';')
 className = className.substring(1, className.length() - 1);
 // --
 }
 // --
}

Test Case 1 EvoSuite
@Test(timeout = 4000)
public void test118() throws Throwable {
 String string0 = TestUtils.getShortClassName("L2");
 assertEquals("L2[]", string0);
}

Test Case 2 ChatGPT-3.5
@Test
public void testGetShortClassName() {
 // Test when className is null
 String result1 = TestUtils.getShortClassName(null);
 assertEquals("", result1);

 String result4 = TestUtils.getShortClassName("[Ljava.lang.String;");
 assertEquals("String[]", result4);
}

Test Case 3 ChatGPT-3.5 + Repair
@Test
public void testGetShortClassName() {
 // Test when className is null
 String result1 = TestUtils.getShortClassName((String) null);
 assertEquals("", result1);

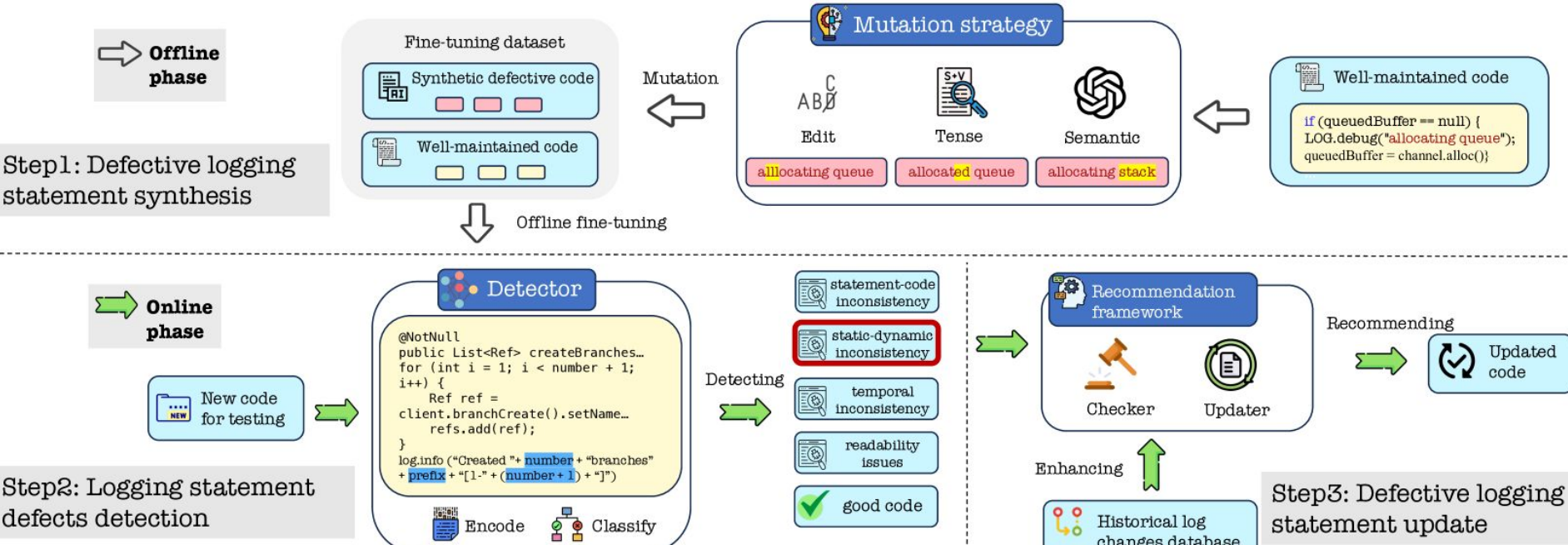
 String result4 = TestUtils.getShortClassName("[Ljava.lang.String;");
 assertEquals("String[]", result4);
}
```



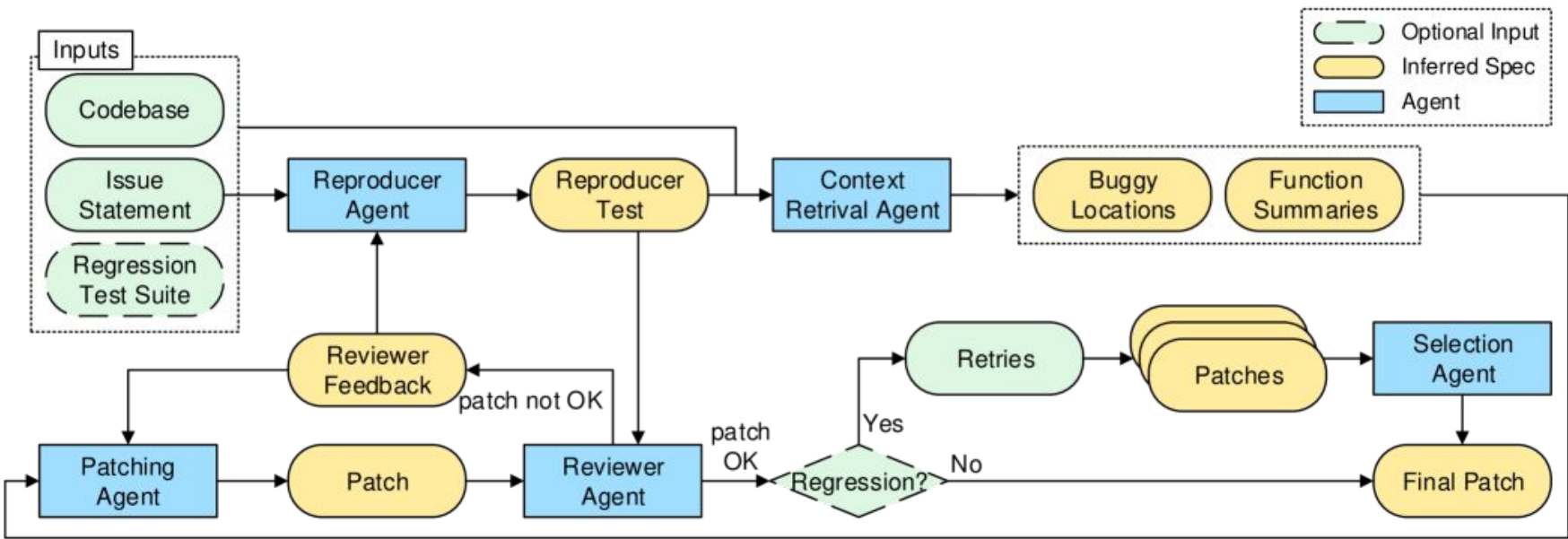
概要: ログステートメントは、ソフトウェアの動作を記録しますが、誤解を招くログが保守を複雑にするため LogFixerでは、合成された欠陥ログを使い、類似性に基づく 4分類器を作成。この分類器は、ログステートメントを評価し、修正が必要かどうかを判断します。次に、過去のログ修正履歴を使い、LLMを使用して修正の提案を行います。

### 技術や手法の詳細説明

LogFixerの手法については以下になります。



概要: LLMを使用した自動プログラム改善手法 SpecRoverはプロジェクトの構造と動作から意図を抽出、仕様を推論、その仕様を基にパッチを生成し、そのパッチの信頼性を確認するためにレビューエージェントを使い、パッチの品質を高めるために、反復的な仕様推論とレビューを行います。



### 技術や手法:

SpecRoverのフローは以下になっています

概要: LLMと対話しながらユーザーの出力の好み追加してタスクに応じたプロンプトを作成する。Conversational Prompt Engineering (CPE)を提案。ユーザーが提供するラベルなしデータを基に質問を生成して、初期の指示を作り、その後、ユーザーのフィードバックを基にさらにプロンプトを洗練させます。

## Conversational Prompt Engineering (CPE)の手法

### 1. プロンプト生成のプロセス

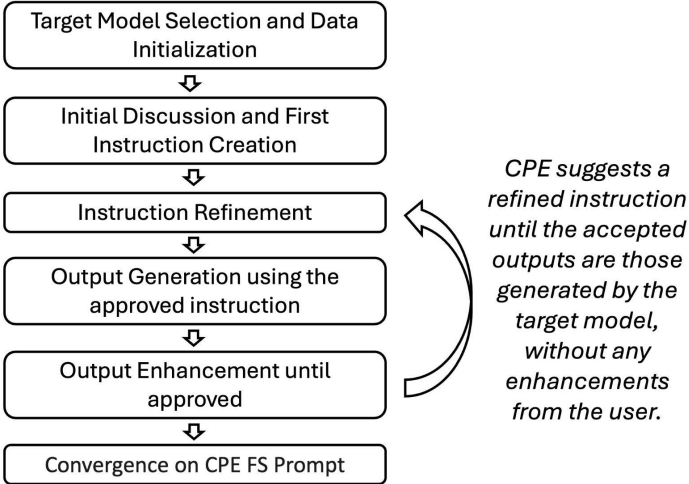
### 1.1 ユーザーからのデータ提供

CPEのプロセスは、ユーザーがラベルなしのデータを提供するところから始まります。このデータは、後続のプロンプト生成に利用される重要な材料です。例えば、ユーザーが映画のレビューを要約したい場合、3つの映画レビューのサンプルを CPEにアップロードします。

### 1.2 データ分析

提供されたラベルなしデータは、CPEによって分析されます。CPEはこのデータを用いて、ユーザーのタスクに関連する要素を特定します。例えば、映画レビューの場合、レビューの中でプロット(物語の概要)やレビューアの意見など、要約に重要な情報が含まれているかを分析します。

### 1.3 データ駆動型の質問の生成



概要: RAGEvalは、シナリオ特化型の評価データセットを生成するフレームワークで、スキーマを使い、縦割りドメインごとにドキュメントを生成、これを基に質問応答ペアを作成し、Completeness、Hallucination、Irrelevanceという指標を使用し応答を評価します

## 技術と手法

### 1. スキーマ要約 (Schema Summary)

**概要**: ドメイン固有のシナリオでは、テキストは通常、共通の知識フレームワークに従います。このフレームワークをスキーマと呼び、ドメイン固有の知識を体系的にまとめたものです。RAGEvalでは、少数の種ドキュメントを分析して、このスキーマを抽出します。

**手順**:

- **ドキュメントの選定**: ドメイン固有のいくつかのドキュメントを種として選びます。
- **LLMを使用した帰納推論**: 選定した種ドキュメントを用いて、LLMを使用し、ドメインに特有の情報(例えば、組織名、イベント、日時、場所など)を要約します。
- **スキーマの確立**: 上記の帰納推論の結果をまとめ、ドキュメントの基本構造を定義するスキーマを作成します。このスキーマは、そのドメインに特有の重要な情報を網羅することを目指します。

### 2. ドキュメント生成 (Document Generation)

概要: LLMとLLMベースエージェントの能力を明確に区別し、 SEにおける要件工学、コード生成、自律的意思決定、設計、テスト生成、保守という 6つの主要トピックについてどのように使用されているかを調査  
それぞれのトピックにおいて、 LLMは主に自動化と効率化を支援し、 LLMベースエージェントはその能力をさらに拡張し、より複雑で自律的なタスクを処理する役割を果たしていました

### 1. 要件工学

### LLMの適用

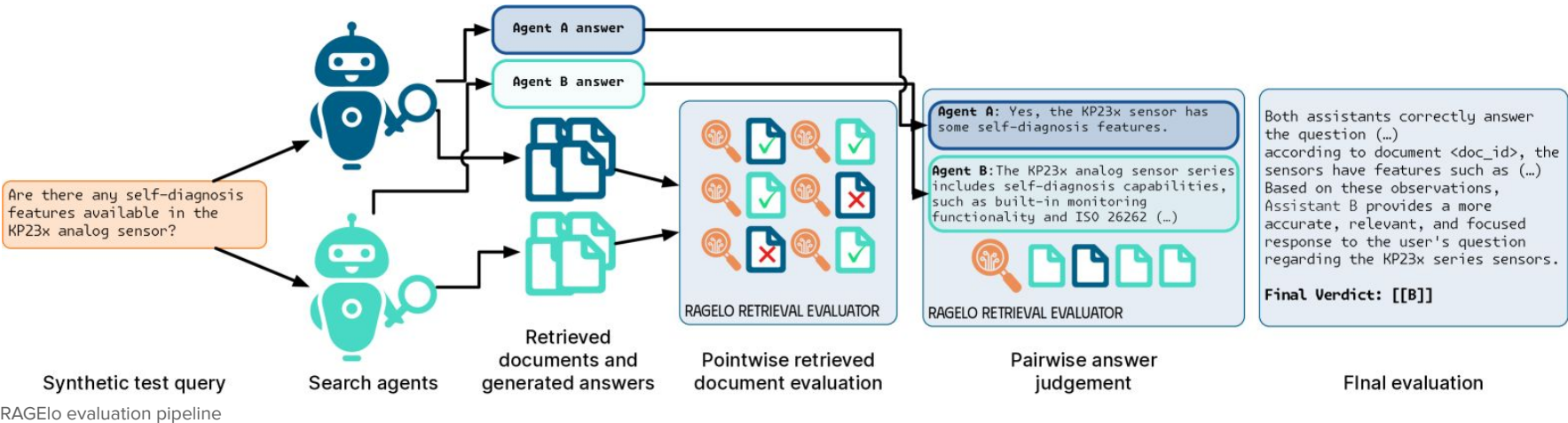
LLM(大規模言語モデル)は要件工学において、要件の抽出、分類、生成、曖昧性の検出、および品質評価に利用されています。

- \*\*要件抽出と分類 \*\*: LLMは自然言語で記述された要件文書から、機能要件と非機能要件を自動的に抽出し、分類することができます。これにより、開発者は要件の整理と分析を効率的に行うことができます。
- \*\*要件生成 \*\*: LLMを使用して、ソフトウェア要件仕様書( SRS)を自動生成することができます。例えば、ユーザーの要求やシステムの目的を自然言語で入力することで、 LLMがそれを解析し、詳細な要件文書を生成します。
- \*\*曖昧性の検出 \*\*: LLMは、要件文書内の曖昧な表現を検出し、明確化のための提案を行います。これにより、誤解を防ぎ、要件の品質を向上させることができます。
- \*\*品質評価 \*\*: LLMは、要件文書の品質を評価し、不足や矛盾点を指摘することができます。これにより、要件の信頼性を向上させることが可能です。

### LLMベースエージェントの適用

概要: RAGによるQAの自動評価で企業内タスクに対するベンチマークの不足に対応するために LLMを活用してユーザーの実際のクエリとドメイン内の文書に基づく合成クエリの大規模データセットを生成し、自動Eloベースの RAGエージェントの異なるバリエーションをランク付けする包括的な RAGElo評価フレームワークを提案。

製品 QAタスクにおいて RAGとRAGFを評価し、RAGEloの評価基準によると RAGFの方が優れた回答をすることがわかりました



## 技術と手法について

### \*\*RAGEloフレームワーク\*\*:

RAGEloは、Eloランキングシステムに基づいた評価ツールキットで、RAGシステムによって生成されたドキュメントや回答を評価するために、LLMをジャッジとして使用します。

概要: Python, C, C++, Java, JavaScriptを対象にリアルタイム脆弱性分析を行う CODEGUARDIANを開発、GPT-4 TurboとGPT-4oは、他の LLMと比較して特に脆弱性検出と CWE分類のタスクで他の LLMよりも優れており、few-shotプロンプトがさらに効果をあげることがわかりました

- \*\*LLMの評価\*\*: 複数のプログラミング言語に対して、 GPT-3.5-Turbo, GPT-4 Turbo, GPT-4o, CodeLlama-7B, CodeLlama-13B, Gemini 1.5 Proの6つの LLMを使用して、脆弱性検出および CWE分類の能力を評価。
- \*\*データセットの構築 \*\*: 5つのプログラミング言語で 370以上の手動で検証された脆弱性を含むデータセットを構築し、脆弱性検出と分類の評価に使用。
- \*\*プロンプト戦略 \*\*: 脆弱性検出と CWE分類のために、複数のシステムプロンプトおよびユーザープロンプトを設計し、 LLMの性能を最適化。
- \*\*CODEGUARDIANの開発 \*\*: VSCodeの拡張機能として、LLMを活用してリアルタイムでコードの脆弱性を分析するツールを開発。
- \*\*GPT-4 TurboとGPT-4oの優位性 \*\*:
  - \*\*脆弱性検出 \*\*では、GPT-4 TurboとGPT-4oが他の LLMを上回るパフォーマンスを示しました。特に、 GPT-4 Turboは誤警報を最小限に抑える点で最も効果的であり、 GPT-4oはCとC++の脆弱性検出で優れたリコール(真の脆弱性を見逃さない率)を示しました。
  - \*\*CWE分類\*\*では、GPT-4oが多言語間で最高の分類スコアを達成し、特に few-shotプロンプトを使用した場合にパフォーマンスが顕著に向上しました。
- \*\*Gemini 1.5 Proの特異なパフォーマンス \*\*:
  - 特にPythonとJavaScriptの脆弱性検出および CWE分類において、Gemini 1.5 Proも強力なパフォーマンスを示しました。
- \*\*few-shot学習の効果 \*\*:
  - CWE分類において、few-shotプロンプトを使用することで、モデルのパフォーマンスが大幅に向上することが確認されました。特に、 GPT-4oはfew-shot設定でリコールが約 3倍に改善されました。
- \*\*CODEGUARDIANの有効性 \*\*:
  - ユーザースタディでは、CODEGUARDIANを使用した開発者が、脆弱性の検出において伝統的な手法と比較して 66%速く、203%高い精度でタスクを完了することが示されました。 CODEGUARDIANは、開発者の作業フローにシームレスに統合でき、より迅速で正確な脆弱性検出を可能にします。

概要: LLMの多様性を改善するための手法として、チェインオブスペシフィケーション( CoS)プロンプティングを提案。

ユーザーが関心を持つ多様性の次元に基づいてテキストを特徴づける構造的多様性という指標を導入。 LLMに構造的特徴を反映した仕様を生成させ、次にその仕様を満たすテキストを生成させることで、多様性を改善することを目指しています。

チェインオブスペシフィケーション( CoS)プロンプティングは、プロンプトエンジニアリングの一種で、 \*\*複数の段階を経て段階的に詳細な仕様を生成し、その仕様を満たすテキストを最終的に生成する \*\*手法です。

### 1. 高次仕様の生成

まず、LLMに対して高次の抽象的な仕様を生成させます。この高次仕様は、生成されるテキストが満たすべき大まかな構造や特徴を定義します。たとえば、詩の生成では「詩のスタイル」や「テーマ」などが高次仕様に該当します。

### 2. 中次仕様の生成

次に、最初に生成された高次仕様を基にして、より詳細な中次の仕様を生成します。これには、具体的な要素や特徴が含まれます。詩の場合、この段階で「感情のトーン」や「イメージ」などが定義されます。

### 3. 低次仕様の生成

さらに詳細な低次の仕様を生成します。この段階では、テキストの具体的な形式や文体に関する仕様が生成され、これに基づいて最終的なテキストが生成されます。例えば、詩のリズムや押韻パターンなどが該当します。



概要: C/C++で書かれたオープンソースソフトウェア( OSS)の脆弱性バージョンのコミットをプログラムスライシングで抽出されたコードを LLMで精査し特定する Vercationを提案

### 1. Vercation: プログラムスライシングと LLMを組み合わせた手法

Vercationは、オープンソースソフトウェア( OSS)の脆弱なバージョンを特定するための手法で、プログラムスライシングと大規模言語モデル( LLM)を組み合わせています。具体的には、以下のプロセスで動作します。

- \*\*プログラムスライシング\*\*を用いて、脆弱性パッチに関連する危険なフロー( vulnerability-related statements)を抽出します。
- \*\*LLM\*\*を活用して、抽出されたステートメントをさらに分析し、脆弱性に関連する部分を精査します。
- 過去のコミットを遡り、脆弱性を導入したコミット( vulnerability-introducing commit, vic)を特定します。
- \*\*セマンティックレベルのコードクローン検出\*\*により、脆弱性を導入したコミットと修正されたコミットの間に存在する脆弱なバージョンを精密に特定します。

Vercationの革新性は、従来の静的解析ツールが持つ限界(シンタックスレベルのコードクローン検出や脆弱性に関係のないコードが混入すること)を克服し、 LLMの力を借りてより精密な脆弱性検出を行える点にあります。

### 2. プログラムスライシング

プログラムスライシングは、ソフトウェアのソースコードから特定の変数やステートメントに関連する部分だけを切り出す技術です。 Vercationでは、以下のようにプログラムスライシングを用いて、脆弱性パッチに関連する「危険なフロー」を抽出します。

概要: LLMが法的助言を提供する際に正確な応答をするため、 BGEを使い、法的記事を検索し法的根拠を提供する ELLAを提案。ユーザーは関連する法的記事を選択し、 LLMにより正確な応答を生成させるために記事をインタラクティブに選ぶ。法的ケースを検索するため、 ELLAは中国の法的ケースデータセットを使い、ユーザーが参考情報を得る。法的根拠を視覚化するため、 BGEのファインチューニングを行う。

### 1. BGEのファインチューニング

**\*\*概要\*\*:**

BGE (Best General Embeddings) は、もともと一般的な文章の埋め込み ( embedding) を生成するためのモデルですが、法的な文脈においても高精度な埋め込みを生成できるように調整 (ファインチューニング) されています。このプロセスにより、法律文書に特有の用語やフレーズ間の類似性をより正確に評価できるようになります。

**\*\*詳細\*\*:**

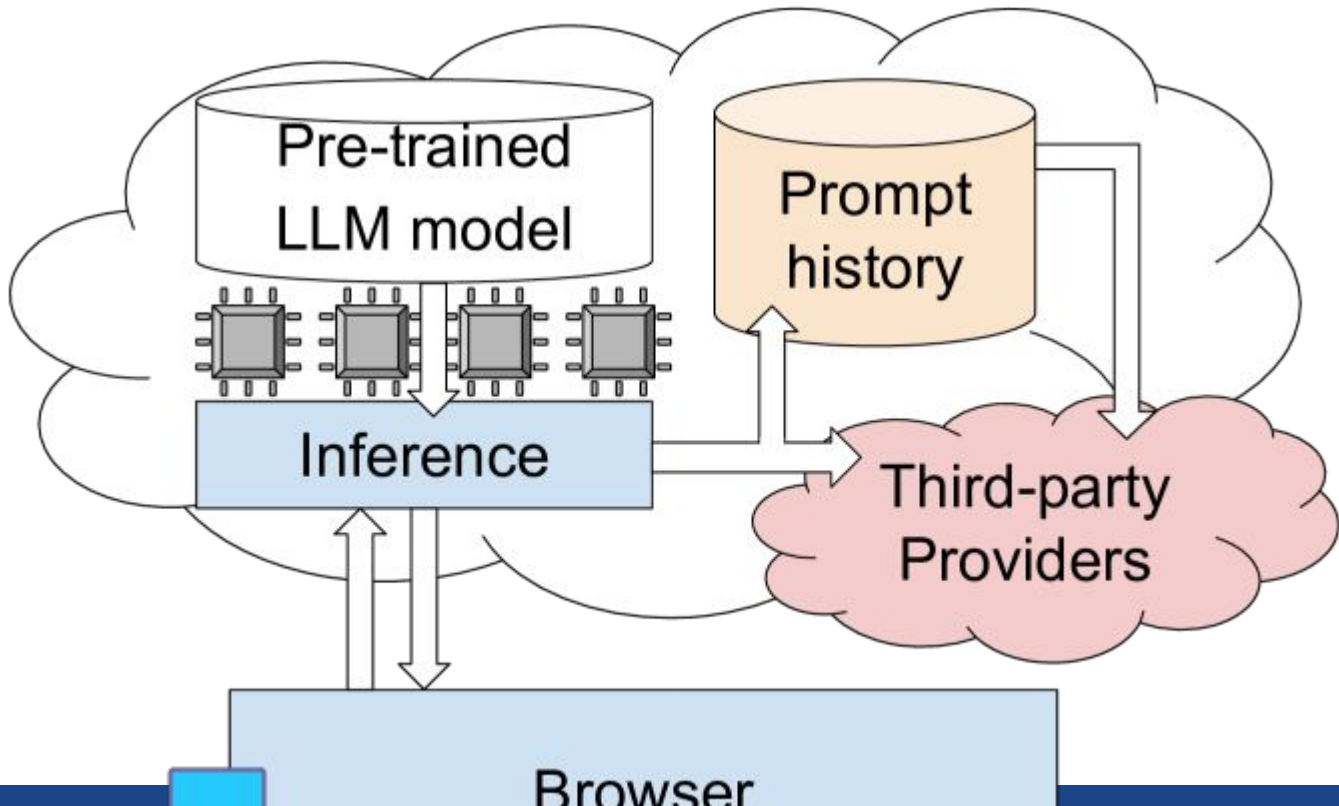
- **\*\*データセットの準備 \*\*:** 法律領域でのファインチューニングのために、法的なデータセットを準備します。具体的には、法的なクエリとそれに関連する法的記事、判例、司法解釈などが含まれたデータが使用されます。

- **\*\*ファインチューニングの目的 \*\*:** BGEモデルが、法律用語や法的文書の特定のフレーズに関する類似性を正確に捉えることを目的としています。たとえば、ある法的概念が別の関連概念とどの程度近い か、またはどの程度異なるかを正確に理解できるようになります。

- **\*\*手法\*\*:** 法的クエリに対して、関連する法的記事を取得し、そのクエリと記事の間の類似性スコアを計算します。そのスコアが高いほど、クエリと記事が密接に関連していると判断されます。この類似性スコアは、文書間の意味的な一致度を反映するもので、法的な文脈に適した回答を導き出すのに役立ちます。

- **\*\*結果\*\*:** ファインチューニングされた BGEモデルは、法的文書の中で特定の用語がどのように使用されているかを理解し、関連する法的根拠を特定する能力が向上します。これにより、 LLMがユーザーの質問に対してより適切で正確な法的助言を提供できるようになります。

概要: ウェブベースの LLMサービスではユーザーのプロンプトデータが保存され、処理され、共有される可能性があります。この対策として Casperというプロンプト消毒技術を提案。ユーザーの入力から敏感な情報を検出し、LLMサービスに送信する前に除去することでプライバシーを保護します。 Casperはユーザーのデバイス上で完全に動作し、オンライン LLMサービスの変更を必要としません。 Casperは、ルールベースのフィルター、機械学習 ( ML) ベースの固有表現認識器、ブラウザベースのローカル LLMトピック識別器からなる三層の消毒メカニズムを持ち、 98.5%の精度で個人識別情報 ( PII) を、89.9%の精度でプライバシーに敏感なトピックをフィルタリングできることを示しました。



概要: LLMが法的助言を提供する際に正確な応答をするため、 BGEを使い、法的記事を検索し法的根拠を提供する ELLAを提案。ユーザーは関連する法的記事を選択し、 LLMにより正確な応答を生成させるために記事をインタラクティブに選ぶ。法的ケースを検索するため、 ELLAは中国の法的ケースデータセットを使い、ユーザーが参考情報を得る。法的根拠を視覚化するため、 BGEのファインチューニングを行う。

### 1. BGEのファインチューニング

**\*\*概要\*\*:**

BGE (Best General Embeddings) は、もともと一般的な文章の埋め込み ( embedding) を生成するためのモデルですが、法的な文脈においても高精度な埋め込みを生成できるように調整 (ファインチューニング) されています。このプロセスにより、法律文書に特有の用語やフレーズ間の類似性をより正確に評価できるようになります。

**\*\*詳細\*\*:**

- **\*\*データセットの準備 \*\*:** 法律領域でのファインチューニングのために、法的なデータセットを準備します。具体的には、法的なクエリとそれに関連する法的記事、判例、司法解釈などが含まれたデータが使用されます。

- **\*\*ファインチューニングの目的 \*\*:** BGEモデルが、法律用語や法的文書の特定のフレーズに関する類似性を正確に捉えることを目的としています。たとえば、ある法的概念が別の関連概念とどの程度近い か、またはどの程度異なるかを正確に理解できるようになります。

- **\*\*手法\*\*:** 法的クエリに対して、関連する法的記事を取得し、そのクエリと記事の間の類似性スコアを計算します。そのスコアが高いほど、クエリと記事が密接に関連していると判断されます。この類似性スコアは、文書間の意味的な一致度を反映するもので、法的な文脈に適した回答を導き出すのに役立ちます。

- **\*\*結果\*\*:** ファインチューニングされた BGEモデルは、法的文書の中で特定の用語がどのように使用されているかを理解し、関連する法的根拠を特定する能力が向上します。これにより、 LLMがユーザーの質問に対してより適切で正確な法的助言を提供できるようになります。

概要: LLMでコード生成でのハイパーパラメータを変更した際の生成結果について調査

具体的には、temperature, top probability (top\_p), frequency penalty, and presence penaltyの4つのハイパーパラメータを調整検証しました。

結果、temperatureが 0.5 以下、top p が 0.75 以下、frequency penaltyが -1 以上 1.5 以下、presence penaltyが -1 以上の設定が最適

### 技術や手法

GPT-3.5 Turbo モデルを使用し、temperature, top probability (top\_p), frequency penalty, and presence penaltyといったハイパーパラメータを調整しました。それぞれのパラメータの影響を確認するために、これらのパラメータの組み合わせを少しずつ変化させ、 Python の 13 のタスクに対して 14,742 のコードセグメントを生成しました。生成されたコードは、以下の 4 つの結果のいずれかに分類されました :

- 1. 正しく機能するコード
- 2. 出力がない
- 3. 実行不可能なコード
- 4. 単体テストに失敗するコード。

概要: LLMを使用してチリ独裁政権に関連する歴史的な文書から自動的に知識グラフを作成する方法を提案

4種類のエンティティと 7種類の関係を持つシンプルなオントロジーを使用して LLMとのやり取りを行いハルシネーションを防止、このアーキテクチャの評価には、文書の小さなサブセットを使用して構築されたゴールドスタンダードグラフを用い、同じ文書セットを処理した際の結果と比較します。

### 先行研究と比べての優位性

- \*\*高精度なエンティティ認識 \*\*: 特に個人の認識精度が高い。
- \*\*シンプルなオントロジー \*\*: ハルシネーションを防ぎつつ、情報の損失を最小限に抑える。
- \*\*自動化の効率 \*\*: 手動でのキュレーションコストを大幅に削減。

### 技術や手法

### 1. オントロジーの設計

オントロジーとは、特定のドメインにおけるエンティティやその関係性を定義するフレームワークです。この論文では、LLMの幻覚を防ぐため、固定されたシンプルなオントロジーを使用しています。このオントロジーでは、以下の 4種類のエンティティと 7種類の関係が定義されています。

**\*\*エンティティの種類 \*\*:**

概要: 対話の文脈を理解し、表現された感情に反応する能力を持つ共感的応答生成を LLMのCoTを利用して感情とその原因を統合するアプローチを提案。事前学習された GPT-2を使用した COMETを利用した原因指向の知識をプロンプトに組み込むことで、生成された応答の一貫性が向上しています。

### 技術や手法

### 1. Cause-Aware Chain-of-Thought (CoT) プロンプト

**概要**:

Chain-of-Thought (CoT) プロンプトは、モデルが複数の推論ステップを経て結論に至る過程を制御する手法です。通常の CoTでは、複雑な推論が必要なタスクにおいて、モデルが思考を整理しながら段階的に結論に至ることが可能になります。この論文では、特に感情とその原因に焦点を当てた「Cause-Aware」なCoTプロンプトを提案しています。

**詳細**:

- **プロンプト設計**:

- 感情原因を重視した CoTプロンプトの設計では、まず対話履歴に基づいて話者の感情を推論し、その感情が何に起因するかを分析するプロセスが導入されています。このプロンプトは、「感情分析」と「原因特定」という2つのステージに分かれています。

- **感情分析**: 最初のステージでは、対話履歴から話者の感情を特定します。たとえば、「ユーザーの感情は何か？」という問いを立て、モデルが対話からその感情を推論します。

- **原因特定**: 次のステージでは、特定された感情の原因を対話履歴から推論します。モデルに対して「感情の原因は何か？」と問い、対話内の特定の発言や状況から原因を導き出します。

# HIAGENT: Hierarchical Working Memory Management for Solving Long-Horizon Agent Tasks with Large Language Model HIAGENT: 長期的なエージェントタスクを解決するための大規模言語モデルによる階層的な作業記憶管理

概要: LLMエージェントの作業記憶の管理として HIAGENTを提案。単一の試行内で蓄積されるイン トライアルメモリ(作業記憶)の階層的管理のために LLMでサブゴールを生成、実行。これに関連する行動・観察ペアを要約しメモリチャンクメモリに保存して使用します

### 先行研究と比べての優位性

- \*\*作業記憶の効率的な管理 \*\*: 従来の方法では全ての行動観察ペアを記憶に保持するが、 HIAGENTはサブゴールを用いて必要な情報のみを保持する。
- \*\*成功率の向上 \*\*: 実験結果により、HIAGENTは成功率を 2倍にし、ステップ数を減少させることが確認された。
- \*\*柔軟なメモリ管理 \*\*: 詳細な軌跡情報を必要に応じて取得できるモジュールを導入。

### 技術や手法の詳細

- \*\*サブゴールベースの作業記憶管理 \*\*:
  - HIAGENTは、まず LLMにサブゴールを生成させ、そのサブゴールを達成するための具体的な行動を生成します。サブゴールが達成されると、その対応する行動 -観察ペアを要約し、作業記憶に追加します。これにより、サブゴールに関連する情報のみを作業記憶に残し、以前のサブゴールに関連する詳細な行動 -観察ペアは削除されます。
- \*\*観察の要約 \*\*:
  - サブゴールが達成されると、観察を要約し、要約された観察情報のみを作業記憶に保存します。この要約は、現在のサブゴールが達成されたかどうかの判断に基づいて行われ、将来のサブゴール生成の際に有益な情報として機能します。
- \*\*トラジェクトリの取得 \*\*:
  - 必要に応じて、過去のサブゴールの詳細な行動 -観察ペアを取得できるモジュールを導入しています。これにより、特定のサブゴールが失敗した場合や新しいチャレンジに直面した場合に、過去の成功経



概要: RAGで幻覚の発生頻度を減らせるかを調査、段落ごとに計画に基づいた検索クエリを生成して検索を行うことで長文テキスト生成タスクにおいて、計画に基づく検索と生成が、より情報量が豊富で、出典に基づく応答を生成できる結果になりました

### 技術や手法

### 1. 幻覚の定義と問題点の整理

**\*\*幻覚\*\***とは、言語モデルが生成するテキストが文法的に正しく、文脈的にも自然に見えるものの、その内容が事実と反している問題を指します。具体的には、モデルが適切な知識を持っていない、もしくはコンテキスト内の情報が不足している場合に、既存の知識や文脈から推測されるテキストを生成します。この結果として、事実とは異なる内容が含まれるテキスト(幻覚)が生成され、特に信頼性が求められる応用において問題となります。

幻覚の発生は、以下のような状況で特に顕著になります:

- **\*\*稀少性 (Rarity)\*\***: モデルが訓練データであまり見かけなかった情報に対して生成を行う場合。
- **\*\*最新性 (Recency)\*\***: モデルの知識が訓練後に起こった新しい事実に関する情報がない場合。
- **\*\*ドメインの異質性 (Domain shift)\*\***: モデルが特定のドメインに訓練されていない、またはそのドメイン外の情報を扱う場合。

これらの幻覚を抑制するために、検索メカニズムを活用して、モデルに必要な知識をコンテキストとして提供するアプローチが考えられています。

# Appendix

---