令和6年度茨城大学工学部情報工学科卒業研究論文

RAGにおける
仮想文書生成に特化したSLMの構築

2025年2月4日

学籍番号:21t4014n

大内元樹

茨城大学

工学部情報工学科

指導教員 新納浩幸教授

令和6年度茨城大学工学部情報工学科卒業研究論文

RAGにおける
仮想文書生成に特化したSLMの構築

著者 大内元樹 (21t4014n)

指導教員 新納浩幸教授

論文要旨

近年, 大規模言語モデルは事前学習されたデータを基に自然な対話をすることが可能だが, 最新の情報や限定的な内容を扱うには限界がある.RAG は言語モデルが持つ知識の限界を克服し, 外部の情報源を活用してより正確な応答を生成する技術である.HyDE は RAG の拡張技術であり,RAG がクエリから埋め込みベクトルを作成するのに対し,LLM に生成させた仮想文書から埋め込みベクトルを作成する. これにより HyDE はノイズ低減や検索精度の向上を可能にしたが, より多くの計算資源を必要とするという問題が存在する.

本研究ではこの問題を低減するため, 複数の小規模言語モデル（SLM）を活用したハイブリッド検索の有効性を検証した. 一般にハイブリッド検索ではキーワード検索とベクトル検索の両方が用いられることが多いが, 実験では複数の SLM が出力させた仮想文書を基とする検索を用いた. また, ランク融合をするにあたって複数のアルゴリズムが存在し, 異なるランク融合アルゴリズムが検索精度に与える影響を考察した.

評価実験では, 仮想文書生成用の 1 つの LLM と 3 つの SLM を用意し,HyDE を実装した. また, それぞれの HyDE において,RAG の性能評価用日本語データセットである JQaRA より検索精度を調査・比較した. 結果, 複数の SLM を用いたハイブリッド検索では,LLM を用いたものと同程度の検索精度を発揮することを確認した. また, ランク融合アルゴリズムである CombMNZ を活用した検索で高いスコアを記録した.CombMNZ は文書の検索頻度を重要視したスコアベースのアルゴリズムであり, 本実験において複数の検索器で検索された文書の重要性を確認することができた. また, 一つの HyDE が検索する文書数や, ランク融合アルゴリズムにおけるハイパパラメータを調整による更なる精度改善の余地があり, これを今後の課題とした.

目 次

第1章　序論

1.1　背景

近年, 自然言語処理分野において, 大規模言語モデル（LLM）の発展が目覚ましく続いている. 中でも RAG（Retrieval Argumented Generation）は, クエリをベクトル空間に埋め込み, 外部データベースに対して検索をかけることで正確な応答を生成する技術である.

しかし,RAG における検索精度の向上は依然として課題であり, 特に入力クエリと取得される文書との関連性を高めるための改善が求められている. これに対して,HyDE（Hypothetical Document Embeddings）を用いることで, より関連性の高い検索結果を得る手法が提案されている.HyDE はクエリでなく仮想文書を埋め込みとして検索するため,RAG に比べノイズ低減や, 検索精度の向上といったメリットがある.

1.2　目的

HyDE の問題点として,RAG に比べ「仮想文書の生成」という追加のステップが存在することから, 計算コストが大きいことが挙げられる. また, 高品質の仮想文書を生成するためには大規模な LLM を使用する必要があり, 運用コストの増大や応答速度の低下が懸念される. 本研究では,HyDE における必要な計算資源を低減させるために, 複数の小規模言語モデル（SLM）を組み合わせたハイブリッド検索の有効性を検証することを目的とする.

またハイブリッド検索の実装に際し, 異なるランク融合アルゴリズム（CombSUM, CombMNZ, Borda, RRF）を適用し, それぞれの手法が検索精度に与える影響についても考察する.

1.3　構成

本論文の構成は以下の通りである.

まず本論文を理解するにあたって必要となる知識を示す（第 2 章）. 次に実験の提案手法を示したのち（第 3 章）, 実験に使用したモデル・データセットおよび結果・考察を示す（第 4 章）. 最後に結論を示す（第 5 章）.

第2章　事前知識

第 2 章では, 本論文を理解するにあたり必要となる基礎的な知識について述べる.

2.1　言語モデル

2.1.1　n-gram モデル

言語モデルとは, ある単語の次に続く単語を予測するモデルのことである. シンプルな例として n-gram モデルを紹介する.

n-gram モデルでは, 連続する n 単語から次の単語を予測する. すなわち, ある単語 w_i を予測する際には直前 n 単語 $w_{i-n} \cdots w_{i-1}$ を入力とし,

$$w_i = \underset{w_i}{\mathrm{argmax}} P(w_i | w_{i-n} \cdots w_{i-1}) \tag{2.1}$$

を求める.

n-gram モデルの特性上,n 単語以上離れた部分の文脈理解はできないため, 長距離での入力における単語予測は難しい.

2.1.2　RNN

n-gram モデルの登場後, 可変長・長距離の文脈を考慮できる RNN（Recurrent Neural Network）が登場した. これらは状態の更新を同一関数で行う再帰型のモデルであり,n-gram モデルと比較して, より長距離の文脈依存関係を考慮することが可能となった.

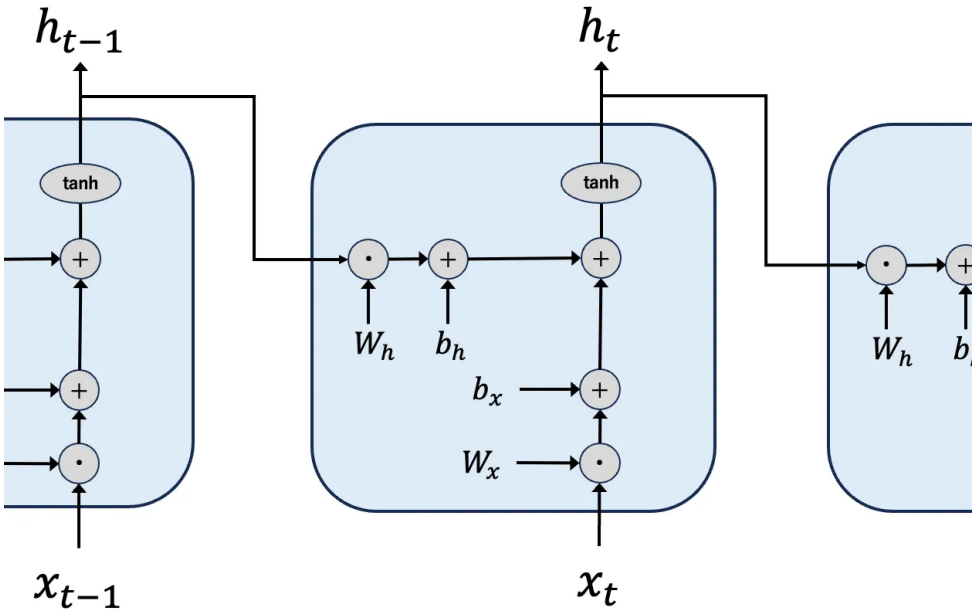

図 2.1: RNN

2.1.3 LSTM

1997 年に登場した LSTM（Long-Short Term Memory）では,3 種類のゲート機構を追加し長期記憶の伝達具合を調整することにより,RNN に比べより長期の文脈依存関係を考慮することが可能となった.

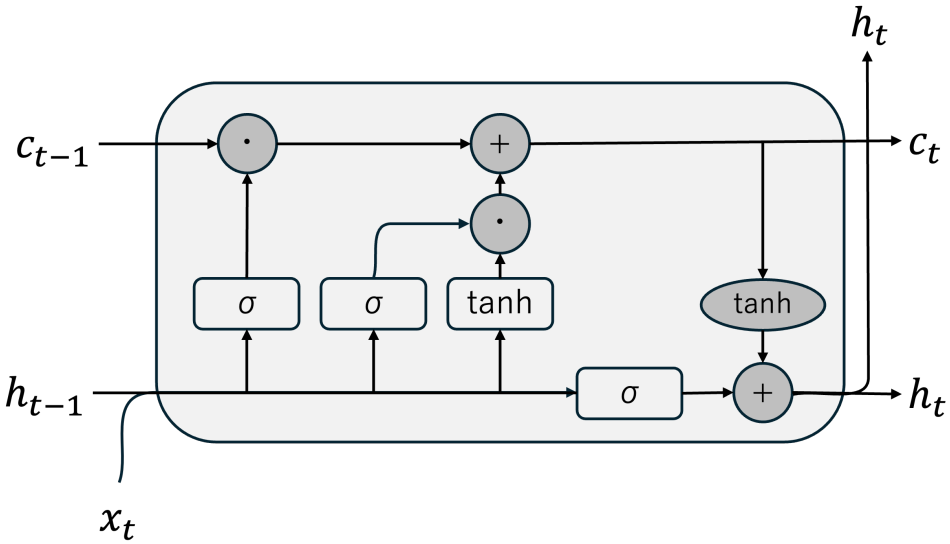

図 2.2: LSTM

2.1.4 Transformer

2017 年に Vaswani によって提案された Transformer [1] は自然言語処理分野におけるブレイクスルーを呼び, 現在の主要な言語モデルの汎用的なアーキテクチャとなっている.RNN・LSTM ブロックのような逐次処理が必要なモジュールに変わり Attention 機構が採用され, 並列処理やより長期の記憶能力が備わった.

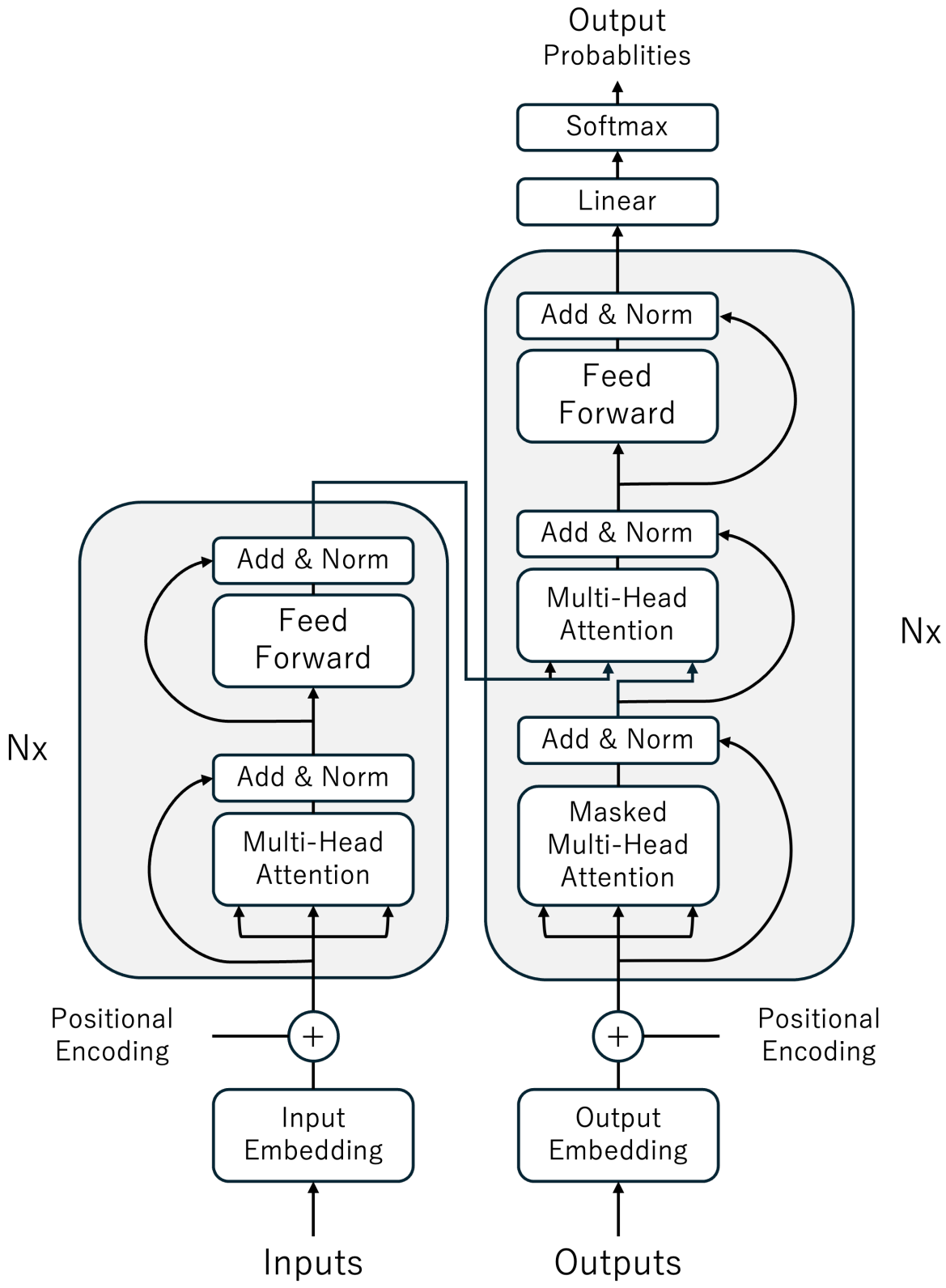

図 2.3: Transformer

2.2 埋め込み表現

自然言語処理分野において, 言語モデルで文章をそのまま扱うことは難しい. モデルに入力する前段階で文章はトークン化され, 各トークンを何らかの数値に変換する作業 (ベクトル化) が必要である. 文章をベクトル化する技術には様々なものがあり, 最もシンプルなものとして one-hot 表現が挙げられる. 図 2.4 のように,one-hot 表現で構成されるベクトルの次元数は語彙サイズと等しく, 単語と次元が一対一対応しており, 単語に対応する次元のみ値が 1, それ以外が 0 となるように構成される. その性質から one-hot 表現で得られるベクトルは高次元疎なベクトルであり, 言語モデルでこれをそのまま扱うのは難しい.

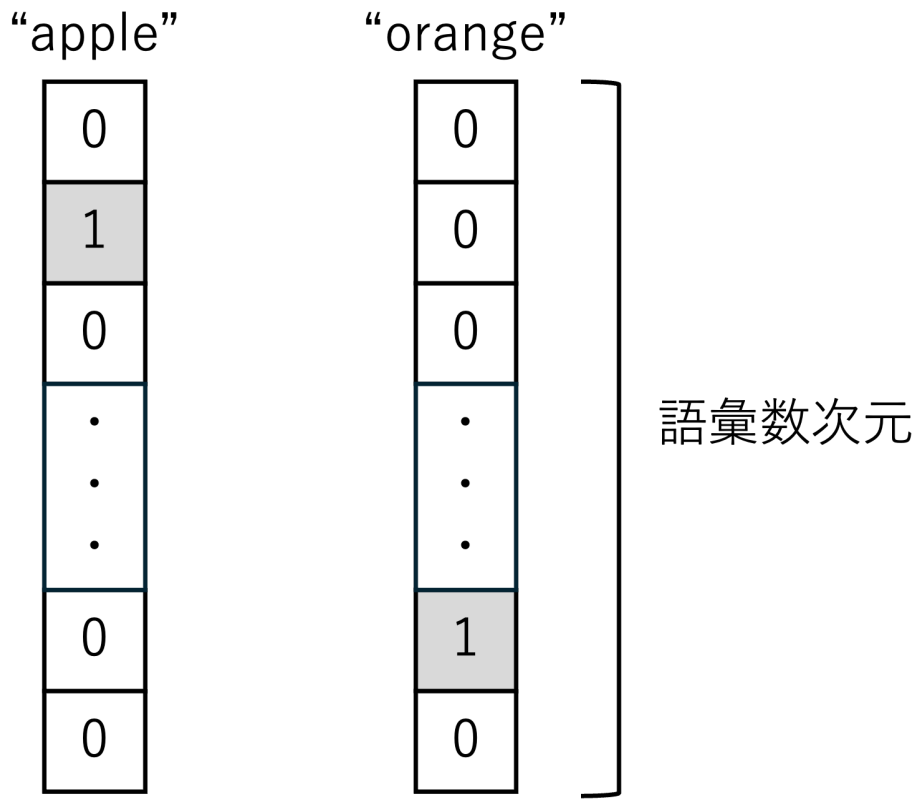

図 2.4: one-hot ベクトル

ディープラーニング技術の発展に伴い, 優れた埋め込み表現の構築方法が確立されてきた. 現在の主な埋め込み表現（分散表現）は図 2.5 に示した通り低次元密なベクトルであり, 単語の意味的な関係まで表現することができている. 埋め込み表現の構築手法として代表的な word2vec について以下に紹介する.

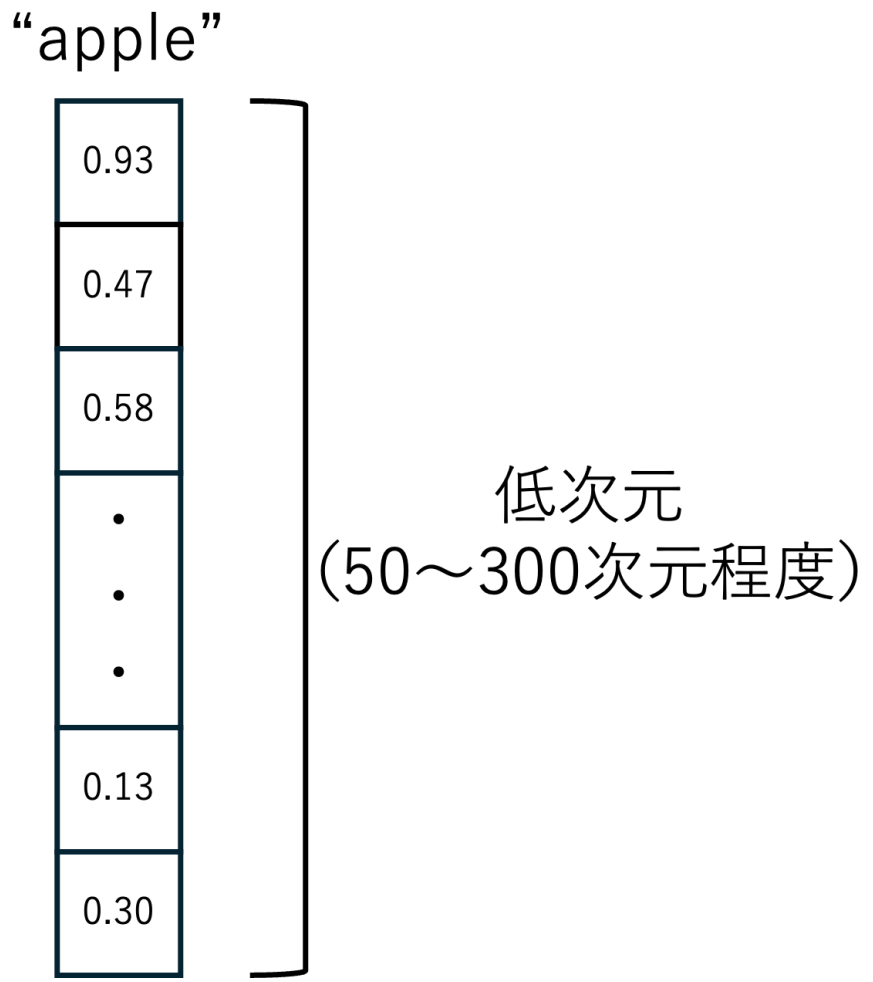

図 2.5: 埋め込み表現（分散表現）

2.2.1 word2vec

word2vec [2] は,2013 年に Tomas Mikolov によって提案された埋め込み表現の構築手法である.CBOW(Continuous Bag-of-Words) と Skip-gram(Continuous Skip-gram) の 2 つの手法が提案され, これから埋め込み表現を獲得することができる.2 つの手法について以下に紹介する.

CBOW

CBOW では周辺の単語から中心の単語を予測するタスクを解く.

例として, 文章「I want XXX play tennis」について XXX(to) の予測問題について, 処理の流れを図 2.6 に則って説明する.

① 入力は各単語の one-hot ベクトルであり, 行列 W_I で次元圧縮する

② 圧縮されたベクトルの平均をとる

③ 行列 W_O で次元を語彙数まで拡張する

④ softmax 関数を通して出力を確率分布に変換する

学習後,W_I が単語の埋め込み表現（各単語の分散表現が横に並んだ行列）として利用される

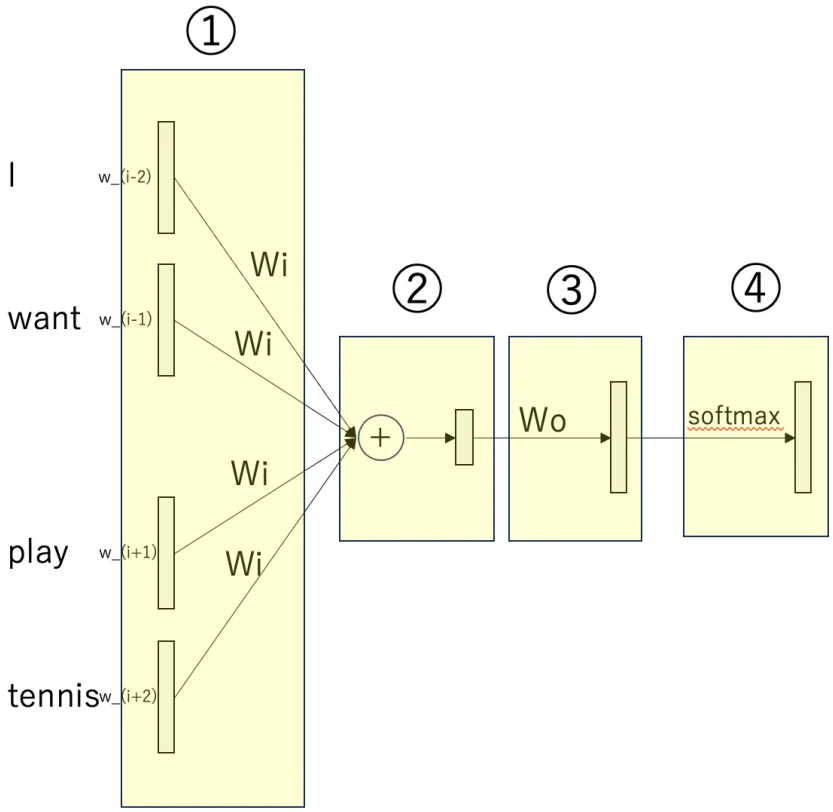

図 2.6: CBOW

Skip-gram

CBOW では周辺後から中心語を予測したのに対して,Skip-gram では図 2.7 のように, 中心語から周辺語を予測する.

① 入力は各単語の one-hot ベクトルであり, 行列 W_I で次元圧縮する

② 行列 W_O で次元を語彙数まで拡張する

③ softmax 関数を通して出力を確率分布に変換する

CBOW と同様, 学習後に W_I が単語の埋め込み表現（各単語の分散表現が横に並んだ行列）として利用される.

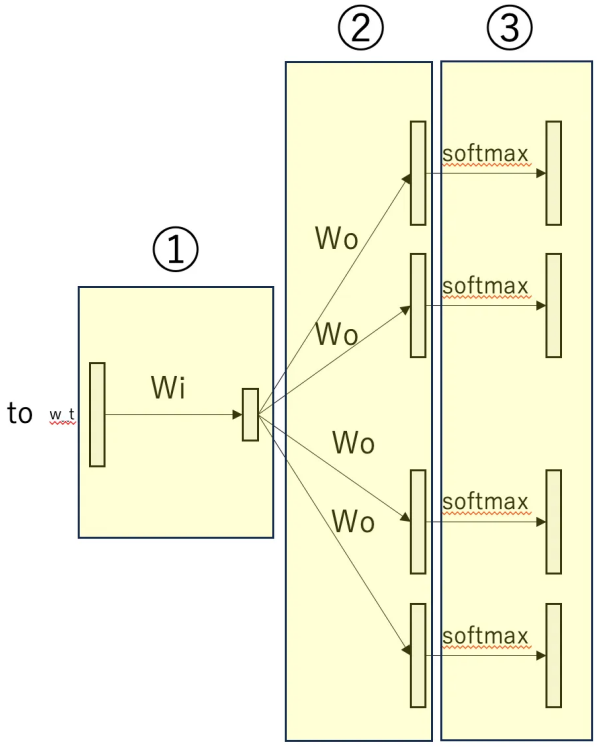

図 2.7: Skip-gram

2.3 RAG

言語モデルの抱える課題の 1 つに, ローカルな情報や最新の情報に弱い, というものがある. 例えば, 社外秘の情報であったり, 最新のニュースに対して正確な回答ができないことが多い. これは, 言語モデルが事前に学習されたデータに基づいて動作しており, 新しい情報や限定的な情報が含まれていない場合があるためである.RAG(Retrieval Argumented Generation) [3] はこの課題を補うことができるアプローチである.

RAG は, クエリに対する回答を生成する前に, クエリと類似する文書を外部データベースから検索する. より具体的には, クエリを埋め込みベクトルに変換し, 外部データベース内の文書の中から類似度の高いものを取得する. 文書の取得後, クエリに取得した文書を付け加え, 言語モデルに質問することによって回答を得る. 一連の流れを図 2.8 に示す.

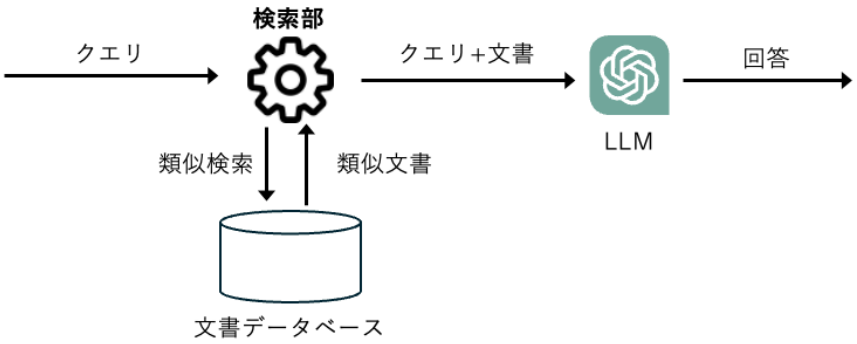

図 2.8: RAG

このプロセスを経て,RAG は言語モデルの学習に含まれていない, 限定的な情報に対しても正確な回答をすることが可能となる.

2.4 HyDE

HyDE(Hypothetical Document Embeddings) [4] は,RAG 分野における Retriever の改善手法である. 一般的な Retriever では外部データベースから情報を取得する際に, 元の入力から埋め込みベクトルを作成していた. しかし, クエリと外部データベース内の文書が類似しているとは限らない. したがって HyDE では, 言語モデルに回答とする仮想文書を生成させ, これを元に埋め込みベクトルを作成する.HyDE における入力から出力までの流れを図 2.9 に示す.

HyDE により, 類似した文書間で検索を行うことができるが, その性質により抱える問題として次のようなものがある.

まず, 仮想文書の生成作業に時間がかかってしまう点である. 一般的な Retriever にはこのプロセスが無いため, 数千・数万以上のクエリを処理する際はその差が大きく現れる.

また, 仮想文書生成用の言語モデルが必要なため, 空間的資源も必要である. 安定した文書生成を行うためには相応規模の言語モデルが必要であり,GPU の並列処理能力による推論の効率化が不可欠である.

以上のように, 一般的な検索拡張生成技術に比べ,HyDE では時間的・空間的なコストが発生しやすい.

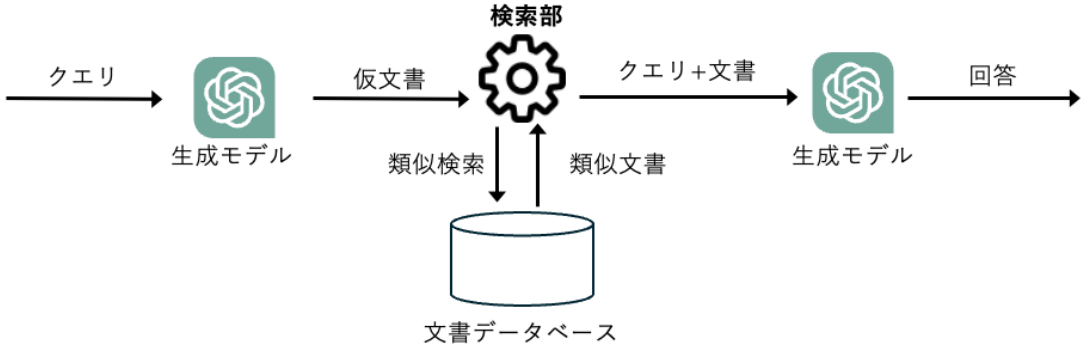

図 2.9: HyDE

2.5 ハイブリッド検索

RAG の精度改善手法の 1 つに, ハイブリッド検索がある. ハイブリッド検索は, 図 2.10 のように, 様々な検索結果をランク融合アルゴリズムによって組み合わせ, 検索結果を改善する手法である.

主要なランク融合アルゴリズムについて, 以下に紹介する.

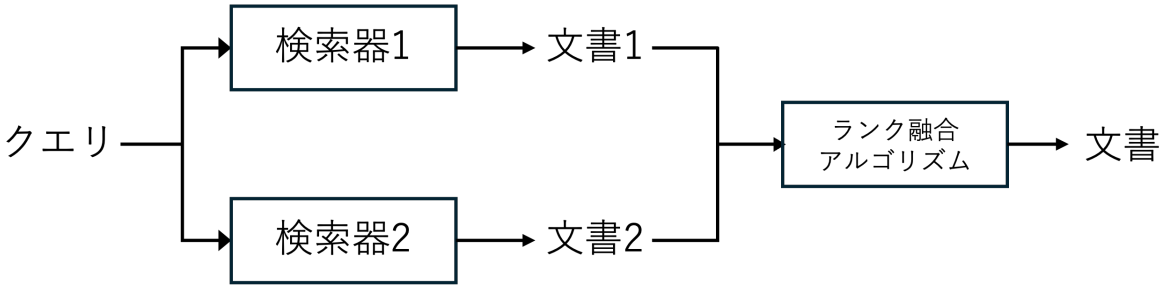

図 2.10: HyDE

2.5.1 CombSUM

CombSUM [5] は Fox らによって提案されたアルゴリズムで, 文書全体の集合を D, 検索器 i による文書 d のスコアを $S_i(d)$ とすると, 以下の式 2.2 で定義される.

$$S_{CombSUM}(d) = \sum_{d \in D} S_i(d) \tag{2.2}$$

したがって,CombMNZ アルゴリズムにおいて, ある文書におけるスコアは, シンプルに全ての検索器によるスコアの和である.

2.5.2 CombMNZ

CombMNZ も CombSUM と同様,Fox らによって提案されたアルゴリズムである. 文書 d を取得した検索器の数を $F(d)$ として, 以下の式 2.3 で定義される.

$$S_{CombMNZ}(d) = \sum_{d \in D} F(d) \cdot S_i(d) \tag{2.3}$$

ある文書を取得した検索器の数が重みとなることによって,CombSUM と比べ, 何度も検索される文書が最終的に選ばれやすくなる.

2.5.3 Borda

Borda(Borda count) は,1770 年に数学者の Borda によって考案されたアルゴリズムである. 文書の数を n とし, 検索器 i における文書 d の順位を $R_i(D)$ とすると, 以下の式 2.4 で定義される.

$$S_{Borda}(d) = \sum_{d \in D} (n - R_i(d)) \tag{2.4}$$

2.5.4 RRF

RRF は Cormack ら [6] によって提案されたアルゴリズムである.k を自由調節可能な値として, 以下の式 2.5 で定義される.

$$S_{RRF}(d) = \sum_{d \in D} \frac{1}{k + R(d)} \tag{2.5}$$

k の値を大きくすればするほど, 下位の文書がランキングに与える影響を大きくすることができる.

2.6 量子化

機械学習モデルを利用する際の現実的な課題として, モデルサイズの大きさによるメモリ不足や推論速度の低下が挙げられる. モデルの量子化はこれらの問題を低減することが

できる手法である. 一般にモデル内での計算は float32 の精度で行われるが, 量子化することにより, より低い精度での計算に変換される. 例えば 8bit 量子化では,float32 で計算していたところを int8 で計算するので, 単純にサイズが 1/4 に節約される. 量子化により空間的・時間的な計算資源は節約できる一方で, 丸めによる誤差の発生により精度劣化が発生するという面もある.

第3章　提案手法

以下のステップで, 各種 LLM について HyDE を実装していく.

1. 質問文に対し,LLM に仮想文書を生成させる.

2. 仮想文書と類似する文書（実文書）を, 外部データベースから検索する.

3. 質問文と実文書をまとめてクエリとし,gpt-4o に回答させる.

本実験では, 最終出力の精度ではなく, 優れた実文書が検索できているかどうかに着目し, 各種 HyDE での検索結果と, それらを用いたハイブリッド検索の結果について比較を行う.

第4章　評価実験

4.1　使用モデル

本研究では以下の 4 つのモデルを使用した.

- ChatGPT-4o
- microsoft/Phi-3-mini-128k-instruct
- llm-jp/llm-jp-3-3.7b-instruct
- google/gemma-2-2b-it

いずれのモデルについても,Hugging Face で公開されているものを利用した. モデルサイズの観点から,ChatGPT-4o を除く 3 つの言語モデルについて, 以後「SLM」とも表記する.

各言語モデルについて, 以下に簡単に紹介する.

4.1.1　ChatGPT-4o

ChatGPT-4o は,2024 年に OpenAI 社が公開した, 多言語対応の大規模言語モデルである. マルチモーダル処理に対応しており, 人間らしさ重視の対話が可能となっている.

4.1.2　microsoft/Phi-3-mini-128k-instruct

Phi-3-mini-128k-instruct は,microsoft 社によって公開された, 約 3.8B サイズのパラメータを持つ instruct モデルである.128K のトークン長に対応可能である.

4.1.3 llm-jp/llm-jp-3-3.7b-instruct

llm-jp-3-3.7b-instruct は, 約 3.7B サイズのパラメータを持つ, 国立情報学研究所 大規模言語モデル開発センターによって公開された instruct モデルである. 日本語と英語中心のコーパスで学習されているため, 日本語タスクに対して十分な性能を発揮することができる.

4.1.4 google/gemma-2-2b-it

gemma-2-2b-it は,google 社によって公開された, 約 2.6B サイズのパラメータをもつ instruct モデルである. 同じく google 社が公開する Gemini と類似した技術で構築されており, 質疑応答や推論, 要約など, 幅広いタスクに対応することができる.

4.2 使用データセット

本研究では,Yuichi Tateno 氏によって公開された,JQaRA(Japanese Question Answering with Retrieval Augmentation) データセットを利用した.JQaRA は RAG の性能評価を目的に構築された日本語質疑応答データセットである. 表 4.1 の通り,1 つの質問に対し 1 つの答えと 100 文の Wikipedia 文データが紐づいているという特徴を持つ.100 文の Wikipedia 文データは,Wikipedia 全文を 400 文字以内にチャンク分割したデータから抽出された, 質問文に類似する上位 100 文のデータである.

表 4.1: JQaRA データセットの構成

質問文	正解文	Wikipedia 文
質問文 1	正解文 1	Wikipedia 文 1-1 ⋮ Wikipedia 文 1-100
質問文 2	正解文 2	Wikipedia 文 1 ⋮ Wikipedia 文 2-100
⋮	⋮	⋮
質問文 1670	正解文 1680	Wikipedia 文 1 ⋮ Wikipedia 文 1670-100

4.3 評価指標

RAG の性能は検索器の性能に大きく依存する. したがって, 検索器が優れた情報を検索できているかどうかの評価指標として, 検索結果が JQaRA データセットにおける質問文に紐付いた 100 文の中にどれほどの割合で含まれているかどうかを調査した. 本実験では test データ内の 300 組の質疑応答を抽出し, 性能評価を行った.

4.4 ハイブリッド検索

3 種類のそれぞれの SLM 単体の検索結果から, ハイブリッド検索を行った. ランク融合アルゴリズムには,CombSUM,CombMNZ,Borda,RRF の 4 種類についてそれぞれ比較した. 実際には図 4.1 に示した通り, 以下のステップでハイブリッド検索を行った.

1. 3 つの SLM（検索器）それぞれにおいて, 質問に対し TOP-5 の類似する Wikipedia 文書を取得する.
2. 重複があった場合は取り除く. この段階で, 文書は 5 個〜15 個用意されることになる.
3. 用意した文書全てについて,3 つの検索器それぞれに対するスコアを計算する.

4. 各ランク融合アルゴリズムについてランク融合を行い,TOP-5 のものを獲得文書とする.RRF のハイパーパラメータ k については,$k = 0, 1, 5, 10, 60$ についてそれぞれ検証する.

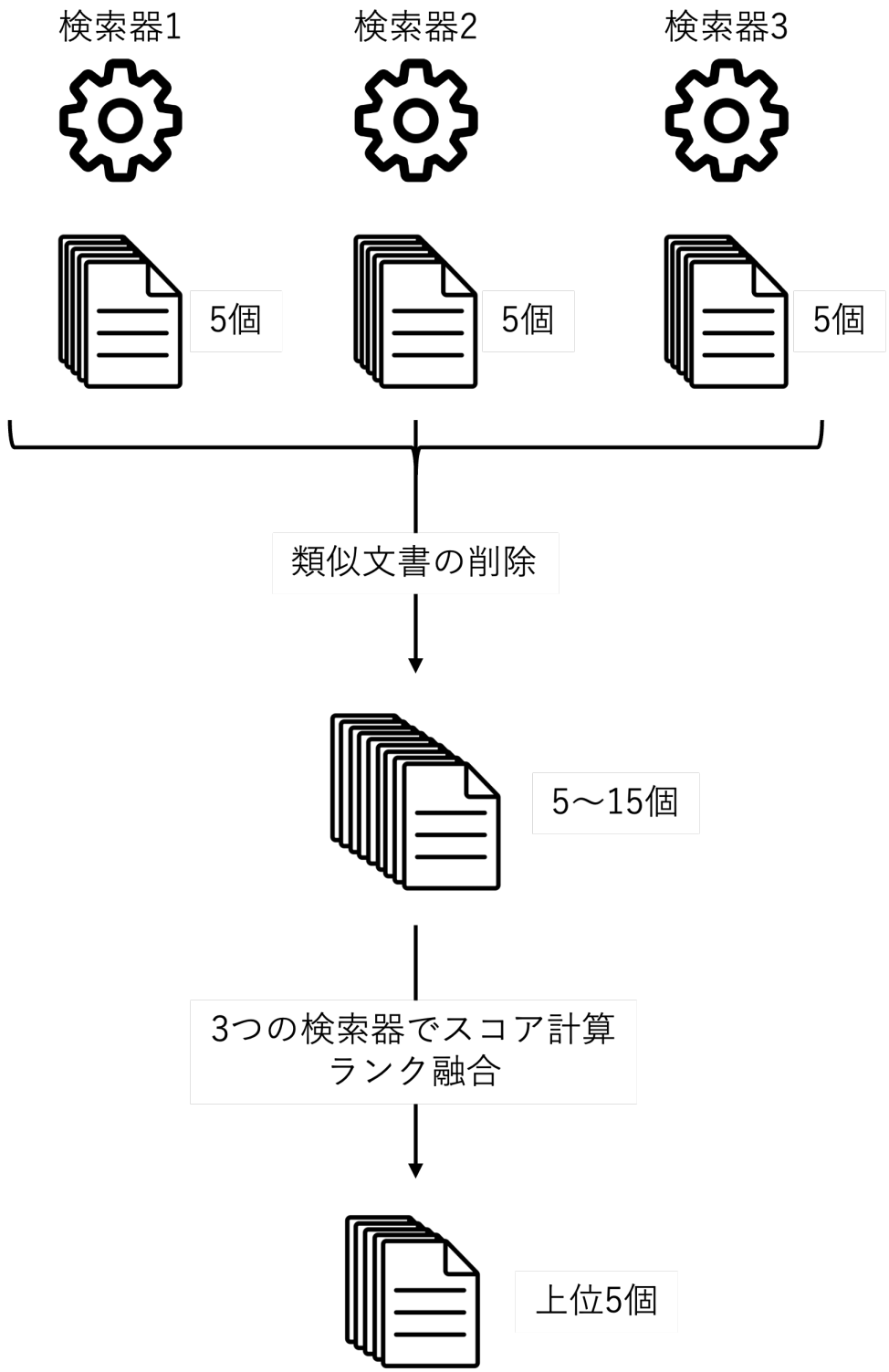

図 4.1: ハイブリッド検索の流れ

また, 各ランク融合アルゴリズムにおけるランク融合の実装を引数の説明と共に以下に示す.

表 4.2: ランク融合における引数の説明

変数名	説明
documents	仮想文書リスト
target_texts	検索対象（Wikipedia 文）
F	検索頻度
D	スコア行列
I	FAISS の検索インデックス
k	RRF のハイパパラメータ

4.4.1 CombSUM

Listing 4.1: CombSUM の実装

```
    def CombSUM(documents, target_texts,F,  D, I, TOP_K):
        score = {}
        for i in range(len(documents)):
            for j in range(len(target_texts)):
                text = target_texts[I[i][j]]
                if text not in score:
                    score[text] = D[i][j]
                else:
                    score[text] += D[i][j]

        sorted_score = sorted(score.items(), key=lambda x:x[1],
         reverse=True)

        res = []
        for item in sorted_score: #sorted_score->(text, score)
            if len(res) >= TOP_K:
                break
            res.append(item[0])
        return res
```

4.4.2 CombMNZ

Listing 4.2: CombMNZ の実装

```
    def CombMNZ(documents, target_texts,F,  D, I, TOP_K):
    score = {}

    for i in range(len(documents)):
        for j in range(len(target_texts)):
            text = target_texts[I[i][j]]
            if text not in score:
                score[text] = D[i][j]
            else:
                score[text] += D[i][j]

    for text in target_texts:
        score[text] *= F[text]

    sorted_score = sorted(score.items(), key=lambda x:x[1],
     reverse=True)

    res = []
    for item in sorted_score:
        if len(res) >= TOP_K:
            break
        res.append(item[0])
    return res
```

4.4.3 Borda

Listing 4.3: Borda の実装

```
    def BORDA(documents, target_texts, D, I, TOP_K):
    score = {}
    m = len(target_texts)
```

```
4
5       for i in range(len(documents)):
6           for j in range(len(target_texts)):
7               text = target_texts[I[i][j]]
8               if text not in score:
9                   score[text] = m-(j+1)
10              else:
11                  score[text] += m-(j+1)
12
13      sorted_score = sorted(score.items(), key=lambda x:x[1],
          reverse=True)
14
15      res = []
16      for item in sorted_score: #sorted_score->(text, score)
17          if len(res) >= TOP_K:
18              break
19          res.append(item[0])
20      return res
```

4.4.4 RRF

Listing 4.4: RRF の実装

```
1       def RRF(documents, target_texts, D, I, k, TOP_K):
2       score = {}
3
4       for i in range(len(documents)):
5           for j in range(len(target_texts)):
6               text = target_texts[I[i][j]]
7               if text not in score:
8                   score[text] = 1 / (k + j+1)
9               else:
10                  score[text] += 1 / (k + j+1)
11
```

```
12      sorted_score = sorted(score.items(), key=lambda x:x[1],
          reverse=True)
13
14      res = []
15      for item in sorted_score: #sorted_score->(text, score)
16          if len(res) >= TOP_K:
17              break
18          res.append(item[0])
19      return res
```

4.5 実験結果

4 種類の言語モデルを用いた HyDE による検索結果と,3 種類の SLM を用いた HyDE によるハイブリッド検索結果について性能評価を行った結果を以下の表 4.3 に示す.

HyDE 単体での検索結果は,$HyDE_{\text{モデル名の略称}}$ で示す.

ハイブリッド検索結果については,$Hybrid_{\text{ランク融合アルゴリズム名}}$ の形で示す.

表 4.3: 検索精度の比較

使用モデル (アルゴリズム)	Score
$HyDE_{GPT}$	0.741
$HyDE_{Phi3}$	0.555
$HyDE_{llmjp}$	0.585
$HyDE_{gemma}$	0.586
$Hybrid_{CombSUM}$	0.731
$Hybrid_{CombMNZ}$	0.737
$Hybrid_{Borda}$	0.719
$Hybrid_{RRF}$(k=0)	0.677
$Hybrid_{RRF}$(k=1)	0.685
$Hybrid_{RRF}$(k=5)	0.709
$Hybrid_{RRF}$(k=10)	0.716
$Hybrid_{RRF}$(k=60)	0.723

表 4.3 より,$HyDE_{GPT}$ 以外において,HyDE 単体の検索器より, ハイブリッド検索を行った時の方が優れた検索結果を得ることができることが分かった. ハイブリッド検索の中では $HyDE_{CombMNZ}$ が最も優れたスコアを記録した.

4.6 考察

表 4.3 より,ChatGPT-4o を用いた HyDE では, 他の SLM を用いた HyDE に比べ突出した検索性能を持っていることが分かる. これは, モデルサイズの観点から, 言語モデル単体での性能に大きな差があるためであると考える.

しかし,3 つの SLM を用いたハイブリッド検索では, 検索性能を大きく向上させることができた. 特に $Hybrid_{CombMNZ}$ では,$HyDE_{GPT}$ に近しいスコアを獲得している.CombMNZ では文書を検索した検索器の数が重みとなることから, 複数の HyDE により検索された文書は大きな有益性を持つことが考えられる.

本実験では,HyDE の検索する実文書の数を $TOPK = 5$ としているが,TOPK の値や,RRF のハイパーパラメータ k,SLM の数を調節することで,$HyDE_{GPT}$ を超えるスコアを獲得できる可能性は十分にあると考える.

また, 本実験では検索部分の性能を評価指標としているが,HyDE 全体での性能評価ができていないため, 今後の課題とする.

第5章　結論

SLM を用いた HyDE による検索結果を複数利用したハイブリッド検索を行うことで, 高性能な HyDE と同等の検索性能を発揮することができた. また, 複数の検索器で検索された文書は, 質問と関連性の高い優れた文書であることが示された.

また,Hybrid 検索の中でも, 使用するランク融合アルゴリズムによって差があり, 高性能な HyDE の検索性能を上回る可能性が示唆された.

謝辞

本論文を作成するにあたり, 指導教員である新納 浩幸教授には, 研究の着想から, 調査, 論文執筆まで多くのご指導をいただきました. 心から感謝申し上げます.

また所属する新納研究室の皆様には多くのご支援をいただきました. お礼申し上げます.

ありがとうございました.

関連図書

[1] A Vaswani. Attention is all you need. Advances in Neural Information Processing Systems, 2017.

[2] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. arXiv preprint arXiv:1301.3781, 2013.

[3] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. Advances in Neural Information Processing Systems, Vol. 33, pp. 9459–9474, 2020.

[4] Devendra Singh Sachan, Mike Lewis, Mandar Joshi, Armen Aghajanyan, Wen tau Yih, Joelle Pineau, and Luke Zettlemoyer. Improving passage retrieval with zero-shot question generation, 2023.

[5] Edward A. Fox and Joseph A. Shaw. Combination of multiple searches. In Text Retrieval Conference, 1993.

[6] Gordon V Cormack, Charles LA Clarke, and Stefan Buettcher. Reciprocal rank fusion outperforms condorcet and individual rank learning methods. In Proceedings of the 32nd international ACM SIGIR conference on Research and development in information retrieval, pp. 758–759, 2009.

付 録A　プログラムリスト

実験に使用したプログラムコードを以下に示す.

Listing A.1: GetHypoDoc_gpt.py

```
#モジュールの用意
import torch
from bert_score import score
from transformers import AutoModelForCausalLM, AutoTokenizer,
  BitsAndBytesConfig, GPTQConfig
import os
from openai import OpenAI
from langchain_community.callbacks import get_openai_callback
from langchain_core.messages import HumanMessage
from langchain_openai import AzureChatOpenAI
from datasets import load_dataset
from datasets.download import DownloadManager
from datasets import load_dataset
from sentence_transformers import SentenceTransformer
from tqdm.notebook import tqdm
import faiss
import pickle
import numpy as np


# GPT-4 準備 (OpenAI)
model_name = "gpt-4o"
client = OpenAI()


```

```
25 # データセット JQaRA
26 ds = load_dataset("hotchpotch/JQaRA")
27 test_data = ds["test"]
28 questions = list(set(ds["test"]["question"]))
29 qa = [[test_data[i]["question"], *test_data[i]["answers"]] for i
     in range(0, len(ds["test"]), 100)]
30
31
32 # 検索準備 wikipedia
33 # wikipedia 日本語データセットのロード
34 wikija_dataset = load_dataset(
35     path="singletongue/wikipedia-utils",
36     name="passages-c400-jawiki-20230403",
37     split="train",
38 )
39 # faiss index のダウンロード
40 dm = DownloadManager()
41 index_local_path = dm.download(
42     f"https://huggingface.co/datasets/hotchpotch/wikipedia-
       passages-jawiki-embeddings/resolve/main/faiss_indexes/
       passages-c400-jawiki-20230403/multilingual-e5-small-passage/
       index_IVF2048_PQ96.faiss"
43 )
44 # faiss index のロード
45 faiss_index = faiss.read_index(index_local_path)
46
47
48 # embeddings へ変換するモデルのロード
49 emb_model = SentenceTransformer("intfloat/multilingual-e5-small")
50 emb_model.max_seq_length = 512
51
52
53 hypo_doc_list = []
54
```

```
55 input_cost_per_1M_tokens = 2.50
56 output_cost_per_1M_tokens = 10.00
57
58 input_tokens = 0
59 output_tokens = 0
60
61 def calc_costs(input_tokens, output_tokens):
62     input_costs = (input_cost_per_1M_tokens / 10**6) *
       input_tokens
63     output_costs = (output_cost_per_1M_tokens / 10**6) *
       output_tokens
64     total_costs = input_costs + output_costs
65     return total_costs
66
67 for i in tqdm(range(300)):
68     system_template = """あなたは幅広い知識を持つ専門家です。質問に答える文章を
       日本語で書いてください。"""
69
70     query, correct_answer = qa[i]
71
72     completion = client.chat.completions.create(
73         model=model_name,
74         messages=[
75             {"role": "system", "content": system_template},
76             {"role": "user", "content": query}
77         ],
78         temperature=0.2,     #大事
79     )
80
81     llms_ans = completion.choices[0].message.content
82     input_tokens += completion.usage.prompt_tokens
83     output_tokens = completion.usage.completion_tokens
84
85     hypo_doc_list.append(llms_ans)
86     print(llms_ans)
```

```
87
88 print("Total␣Cost␣(USD):", calc_costs(input_tokens, output_tokens
     ))
89
90
91 #仮想文書の保存
92 path_w = "data/gpt4o_hypo-ans"
93 with open(path_w , mode='wb') as f:
94     pickle.dump(hypo_doc_list, f)
```

Listing A.2: eval_gpt.py

```
1  #仮想文書のロード
2  path_r = "data/gpt4o_hypo-ans"
3  with open(path_r, mode='br') as f:
4      gpt4o_hypodocs = pickle.load(f)
5
6
7  #ベクトル化関数
8  def to_emb(emb_model, text, prefix="query:␣"):
9      return emb_model.encode([prefix + text], normalize_embeddings
         =True)
10
11 #実文書の検索
12 def get_real_document(query):
13     TOP_K=5
14     emb = to_emb(emb_model, query)
15     scores, indexes = faiss_index.search(emb, TOP_K)
16     texts = []
17     for idx, (id, score) in enumerate(zip(indexes[0], scores[0]))
         :
18         data = wikija_dataset[int(id)]
19         texts.append(data["text"])
20
21     return texts
```

```
22
23
24 #評価
25 cnt = 0
26 for i in tqdm(range(len(gpt4o_hypodocs))):
27     cdoc = test_data["text"][i*100:i*100+100]
28     res = get_real_document(gpt4o_hypodocs[i])
29     for j in range(5):
30         if res[j] in cdoc:
31             cnt += 1
32
33 print("gpt4o", cnt / (5*len(gpt4o_hypodocs)))
```

Listing A.3: GetHypoDoc_slm.py

```
1  import torch
2  from bert_score import score
3  from transformers import AutoModelForCausalLM, AutoTokenizer,
     BitsAndBytesConfig, GPTQConfig
4  import os
5  from openai import OpenAI
6  from langchain_community.callbacks import get_openai_callback
7  from langchain_core.messages import HumanMessage
8  from langchain_openai import AzureChatOpenAI
9  from datasets import load_dataset
10 from datasets.download import DownloadManager
11 from datasets import load_dataset
12 from sentence_transformers import SentenceTransformer
13 from tqdm.notebook import tqdm
14 from huggingface_hub import login
15 from google.colab import userdata
16 import faiss
17 import pickle
18 import numpy as np
19
```

```
20
21 ds = load_dataset("hotchpotch/JQaRA")
22 test_data = ds["test"]
23 questions = list(set(ds["test"]["question"]))
24 qa = [[test_data[i]["question"], *test_data[i]["answers"]] for i
     in range(0, len(ds["test"]), 100)]
25
26
27 wikija_dataset = load_dataset(
28     path="singletongue/wikipedia-utils",
29     name="passages-c400-jawiki-20230403",
30     split="train",
31 )
32
33 dm = DownloadManager()
34 index_local_path = dm.download(
35     f"https://huggingface.co/datasets/hotchpotch/wikipedia-
       passages-jawiki-embeddings/resolve/main/faiss_indexes/
       passages-c400-jawiki-20230403/multilingual-e5-small-passage/
       index_IVF2048_PQ96.faiss"
36 )
37
38
39 faiss_index = faiss.read_index(index_local_path)
40
41
42 emb_model = SentenceTransformer("intfloat/multilingual-e5-small")
43 emb_model.max_seq_length = 512
44
45
46 quantization_config = BitsAndBytesConfig(
47     load_in_4bit=True,
48     #load_in_8bit=True,
49     bnb_4bit_quant_type="nf4",
```

```
    bnb_4bit_compute_dtype=torch.bfloat16
)


tokenizer1 = AutoTokenizer.from_pretrained(
    "microsoft/Phi-3-mini-128k-instruct"
)
model1 = AutoModelForCausalLM.from_pretrained(
    "microsoft/Phi-3-mini-128k-instruct",
    device_map="cuda",
    torch_dtype="auto",
    trust_remote_code=True,
    #quantization_config=quantization_config,
)


tokenizer2 = AutoTokenizer.from_pretrained(
    "llm-jp/llm-jp-3-3.7b-instruct"
)
model2= AutoModelForCausalLM.from_pretrained(
    "llm-jp/llm-jp-3-3.7b-instruct",
    device_map="cuda",
    torch_dtype="auto",
    trust_remote_code=True,
    #quantization_config=quantization_config
)
tokenizer2.chat_template = "{{bos_token}}{% for message in
  messages %}{% if message['role'] == 'user' %}{{ '\n\n### 指
  示:\n' + message['content'] }}{% elif message['role'] == 'system'
  %}{{ message['content'] }}{% elif message['role'] == 'assistant
  ' %}{{ '\n\n### 応
  答:\n' + message['content'] + eos_token }}{% endif %}{% if loop.
  last and add_generation_prompt %}{{ '\n\n### 応
  答:\n' }}{% endif %}{% endfor %}"
tokenizer2.padding_side = "left"

```

```
79
80  tokenizer3 = AutoTokenizer.from_pretrained(
81      "google/gemma-2-2b-it"
82  )
83  model3= AutoModelForCausalLM.from_pretrained(
84      "google/gemma-2-2b-it",
85      device_map="cuda",
86      torch_dtype="auto",
87      trust_remote_code=True,
88      #quantization_config=quantization_config
89  )
90
91
92  def hypo_docs(n, model, tokenizer, batch_size=1):
93      global error_cnt
94
95      system_template = """あなたは幅広い知識を持つ専門家です。質問に答える文章を
          日本語で書いてください。"""
96
97      for start_idx in tqdm(range(0, n, batch_size)):
98      end_idx = min(start_idx+batch_size, n)
99      batch_queries = [qa[i] for i in range(start_idx, end_idx)]
100
101     chat_prompts = []
102     for query, _ in batch_queries:
103         chat = [
104             {"role": "system", "content": system_template},
105             {"role": "user", "content": query},
106         ]
107         """
108         chat = [
109             {"role": "user", "content": system_template + "\n" +
                query}
110         ]
111         """
```

```
112
113         prompt = tokenizer.apply_chat_template(chat, tokenize=
              False, add_generation_prompt=True)
114         chat_prompts.append(prompt)
115
116     if tokenizer.pad_token is None:
117         tokenizer.pad_token = tokenizer.eos_token
118
119     inputs = tokenizer(
120         chat_prompts,
121         padding=True,
122         padding_side=tokenizer.padding_side,
123         add_special_tokens=True,
124         return_tensors="pt",)
125     token_ids = inputs["input_ids"]
126     attention_mask=inputs["attention_mask"]
127
128     with torch.no_grad():
129         output_ids = model.generate(
130             token_ids.to(model.device),
131             attention_mask=attention_mask.to(model.device),
132             do_sample=True,
133             temperature=0.2,
134             repetition_penalty=1.10,
135             max_new_tokens=256,
136             pad_token_id=tokenizer.pad_token_id,
137             bos_token_id=tokenizer.bos_token_id,
138             eos_token_id=tokenizer.eos_token_id
139         )
140
141     for i, output in enumerate(output_ids):
142         llms_ans = tokenizer.decode(output[token_ids.size(1):],
              skip_special_tokens=True)
143         hypo_doc_list.append((start_idx + i, llms_ans))
```

```
144
145
146 hypo_doc_list = [] #(id, hypodoc)
147 error_cnt = 0
148 n = 300
149 hypo_docs(n, model1, tokenizer1, batch_size=16)
150
151
152 path_w = "data/phi3_hypo-ans"
153 with open(path_w , mode='wb') as f:
154     pickle.dump(hypo_doc_list, f)
```

Listing A.4: eval_slm.py

```
1  path_r = "data/phi3_hypo-ans"
2  with open(path_r, mode='br') as f:
3      phi3_hypodocs = pickle.load(f)
4
5  path_r = "data/llm-jp_hypo-ans"
6  with open(path_r, mode='br') as f:
7      llmjp_hypodocs = pickle.load(f)
8
9  path_r = "data/gemma2_hypo-ans"
10 with open(path_r, mode='br') as f:
11     gemma2_hypodocs = pickle.load(f)
12
13 hypo_documents = [[d1[1], d2[1], d3[1]] for d1, d2, d3 in zip(
     phi3_hypodocs, llmjp_hypodocs, gemma2_hypodocs)]
14
15 def CombSUM(documents, target_texts,F,  D, I, TOP_K):
16     score = {}
17     for i in range(len(documents)):
18         for j in range(len(target_texts)):
19             text = target_texts[I[i][j]]
20             if text not in score:
```

```
21                  score[text] = D[i][j]
22              else:
23                  score[text] += D[i][j]
24
25      sorted_score = sorted(score.items(), key=lambda x:x[1],
          reverse=True)
26
27      res = []
28      for item in sorted_score: #sorted_score->(text, score)
29          if len(res) >= TOP_K:
30              break
31          res.append(item[0])
32      return res
33
34
35  def CombMNZ(documents, target_texts,F,  D, I, TOP_K):
36      score = {}
37
38      for i in range(len(documents)):
39          for j in range(len(target_texts)):
40              text = target_texts[I[i][j]]
41              if text not in score:
42                  score[text] = D[i][j]
43              else:
44                  score[text] += D[i][j]
45
46      for text in target_texts:
47          score[text] *= F[text]
48
49      sorted_score = sorted(score.items(), key=lambda x:x[1],
          reverse=True)
50
51      res = []
52      for item in sorted_score:
```

```
53          if len(res) >= TOP_K:
54              break
55          res.append(item[0])
56      return res
57
58
59  def BORDA(documents, target_texts, D, I, TOP_K):
60      score = {}
61      m = len(target_texts)
62
63      for i in range(len(documents)):
64          for j in range(len(target_texts)):
65              text = target_texts[I[i][j]]
66              if text not in score:
67                  score[text] = m-(j+1)
68              else:
69                  score[text] += m-(j+1)
70
71      sorted_score = sorted(score.items(), key=lambda x:x[1],
          reverse=True)
72
73      res = []
74      for item in sorted_score: #sorted_score->(text, score)
75          if len(res) >= TOP_K:
76              break
77          res.append(item[0])
78      return res
79
80
81  def RRF(documents, target_texts, D, I, k, TOP_K):
82      score = {}
83
84      for i in range(len(documents)):
85          for j in range(len(target_texts)):
```

```
86              text = target_texts[I[i][j]]
87              if text not in score:
88                  score[text] = 1 / (k + j+1)
89              else:
90                  score[text] += 1 / (k + j+1)
91
92      sorted_score = sorted(score.items(), key=lambda x:x[1],
          reverse=True)
93
94      res = []
95      for item in sorted_score: #sorted_score->(text, score)
96          if len(res) >= TOP_K:
97              break
98          res.append(item[0])
99      return res
100
101
102 def to_emb(emb_model, text, prefix="query: "):
103     return emb_model.encode([prefix + text], normalize_embeddings
          =True)
104
105 def hybrid_document(documents, TOP_K=5, strategy="CombSUM", RRF_K
      =0):
106     target_texts = set()
107     F = {}
108
109     for i in range(len(documents)):
110         query = documents[i]
111         emb = to_emb(emb_model, query)
112         scores, indexes = faiss_index.search(emb, TOP_K)
113
114         for idx, (id, score) in enumerate(zip(indexes[0], scores
              [0])):
115             data = wikija_dataset[int(id)]
```

```
116             target_texts.add(data["text"])
117             if data["text"] not in F:
118                 F[data["text"]] = 1
119             else:
120                 F[data["text"]] += 1
121
122     index_CombMNZ = faiss.IndexFlatL2(384)
123
124     target_texts = list(target_texts)
125     target_embeds = [e for e in emb_model.encode(target_texts,
          normalize_embeddings=True)]
126     target_embedds_array = np.array(target_embeds).astype('
          float32')
127
128     index_CombMNZ.add(target_embedds_array)
129
130     input_embeds = [e for e in emb_model.encode(documents,
          normalize_embeddings=True)]
131     query = np.array(input_embeds).astype('float32')
132
133     D, I = index_CombMNZ.search(query, len(target_texts))
134
135     D = 1 - (D - D.min()) / (D.max() - D.min())
136
137     if strategy == "CombSUM":
138         res = CombSUM(documents, target_texts, F, D, I, TOP_K)
139     elif strategy == "CombMNZ":
140         res = CombMNZ(documents, target_texts, F, D, I, TOP_K)
141     elif strategy == "BORDA":
142         res = BORDA(documents, target_texts, D, I, TOP_K)
143     elif strategy == "RRF":
144         res = RRF(documents, target_texts, D, I, RRF_K, TOP_K)
145     else:
146         print("error")
```

```
147
148     return res
149
150
151 def get_real_document(query):
152     TOP_K=5
153     emb = to_emb(emb_model, query)
154     scores, indexes = faiss_index.search(emb, TOP_K)
155     texts = []
156     for idx, (id, score) in enumerate(zip(indexes[0], scores[0]))
        :
157         data = wikija_dataset[int(id)]
158         texts.append(data["text"])
159
160     return texts
161
162
163 cnt = 0
164 for i in tqdm(range(len(hypo_documents))):
165     cdoc = test_data["text"][i*100:i*100+100]
166     res = get_real_document(phi3_hypodocs[i][1])
167     for j in range(5):
168         if res[j] in cdoc:
169             cnt += 1
170
171 print("phi-3", cnt / (5*len(hypo_documents)))
172
173
174 cnt = 0
175 for i in tqdm(range(len(hypo_documents))):
176     cdoc = test_data["text"][i*100:i*100+100]
177     res = get_real_document(llmjp_hypodocs[i][1])
178     for j in range(5):
179         if res[j] in cdoc:
```

```
180                 cnt += 1
181
182 print("llm-jp", cnt / (5*len(hypo_documents)))
183
184
185 cnt = 0
186 for i in tqdm(range(len(hypo_documents))):
187     cdoc = test_data["text"][i*100:i*100+100]
188     res = get_real_document(gemma2_hypodocs[i][1])
189     for j in range(5):
190         if res[j] in cdoc:
191             cnt += 1
192
193 print("gemma2", cnt / (5*len(hypo_documents)))
194
195
196 cnt = 0
197 for i in tqdm(range(len(hypo_documents))):
198     cdoc = test_data["text"][i*100:i*100+100]
199     res = hybrid_document(hypo_documents[i], TOP_K=5, strategy="
      BORDA")
200     for j in range(5):
201         if res[j] in cdoc:
202             cnt += 1
203
204 print("hybrid", cnt / (5*len(hypo_documents)))
```