基于 GraphRAG 的菜谱智能推荐系统
——LLM 与语义生成模块技术报告

Project Technical Report

November 2025

Student 张子烨
Student ID 25125234

目录

1 引言

1.1 项目背景

大语言模型（Large Language Model, LLM）在自然语言理解和生成任务上展现出卓越的能力。然而，直接使用 LLM 进行推荐往往面临幻觉（Hallucination）和知识过时等问题。检索增强生成（RAG）技术通过将外部知识库的检索结果作为上下文输入，有效提升了生成内容的准确性和可靠性。

本项目构建了一个基于 GraphRAG 的菜谱推荐系统，其中 LLM 负责根据检索到的相似菜谱生成可解释的推荐理由。这种" 检索 + 生成" 的模式既保证了推荐的准确性，又提供了良好的用户体验。

1.2 本人负责模块

作为 LLM 与语义生成负责人，本人主要负责以下模块的设计与实现：

- **LLM 生成器**（`llm_generator.py`）：封装 LLM API 调用，设计 Prompt 模板，实现回退策略
- **语义向量索引**（`embeddings.py`）：基于 Sentence-Transformers 的菜谱向量化与相似度检索
- **Prompt 工程**：设计面向菜谱推荐场景的提示模板
- **LLM 配置与封装**：基于 OpenAI Responses API，预留 Provider/模型配置参数

1.3 主要贡献

1. 设计并实现了结构化的 Prompt 模板，提升 LLM 生成质量
2. 构建了高效的语义向量索引，支持文本到菜谱的语义检索
3. 实现了优雅的降级策略，确保无 API Key 时系统仍可运行
4. 封装 OpenAI Responses API，统一管理模型与密钥配置

2 相关工作

2.1 大语言模型与推荐系统

近年来，LLM 在推荐系统中的应用受到广泛关注。与传统推荐方法相比，LLM 能够：

- 理解用户的自然语言查询意图
- 生成人类可读的推荐解释
- 融合多种类型的上下文信息

代表性工作包括 ChatRec、RecLLM 等，它们探索了如何将 LLM 作为推荐系统的核心组件。

2.2 Prompt Engineering

Prompt 工程是影响 LLM 输出质量的关键因素。有效的 Prompt 设计需要考虑：

- **任务描述**：明确告知模型扮演的角色和任务目标
- **上下文信息**：提供充分的背景知识
- **输出格式**：指定期望的回答结构
- **示例引导**：通过 Few-shot 示例提升一致性

2.3 Sentence Embeddings

句子嵌入技术将文本映射到稠密向量空间，使得语义相似的文本在向量空间中距离较近。Sentence-BERT 及其后续工作（如 all-MiniLM-L6-v2）通过对比学习优化了句子级别的语义表示，广泛应用于语义搜索、文本聚类等任务。

3 模型架构与数学推导

3.1 语义向量索引架构

语义向量索引模块的核心是将菜谱文本映射到向量空间：

$$\mathbf{v}_i = f_{encoder}(text_i), \quad \mathbf{v}_i \in \mathbb{R}^d \tag{1}$$

其中 $f_{encoder}$ 是预训练的 Sentence Transformer 模型，$d = 384$ 是 all-MiniLM-L6-v2 的输出维度。

3.2 相似度计算

对于归一化后的向量，余弦相似度等价于内积：

$$sim(\mathbf{v}_i, \mathbf{v}_j) = \cos(\mathbf{v}_i, \mathbf{v}_j) = \frac{\mathbf{v}_i \cdot \mathbf{v}_j}{\|\mathbf{v}_i\|\|\mathbf{v}_j\|} = \mathbf{v}_i^T \mathbf{v}_j \tag{2}$$

批量计算相似度可通过矩阵乘法高效实现：

$$\mathbf{S} = \mathbf{V}\mathbf{q}, \quad \mathbf{S} \in \mathbb{R}^n \tag{3}$$

其中 $\mathbf{V} \in \mathbb{R}^{n \times d}$ 是菜谱向量矩阵，$\mathbf{q} \in \mathbb{R}^d$ 是查询向量。

3.3 Top-K 检索

给定相似度分数向量 $\mathbf{S}$，Top-K 检索可表示为：

$$TopK(\mathbf{S}, k) = \arg\max_{|I|=k} \sum_{i \in I} S_i \tag{4}$$

实现中使用 `np.argpartition` 获得 O(n) 复杂度的部分排序。

3.4 Prompt 模板设计

Prompt 模板采用结构化设计：

$$Prompt = Role + UserInput + ReferenceRecipe + CandidateRecipes + Instruction \tag{5}$$

各部分的信息熵分布：

- $Role$：低熵，固定角色描述
- $UserInput$：中熵，用户原始输入
- $ReferenceRecipe$：高熵，参考菜谱详细信息
- $CandidateRecipes$：高熵，候选菜谱列表
- $Instruction$：低熵，输出指令

3.5 LLM 生成概率模型

LLM 生成过程可建模为条件概率分布：

$$P(y|x) = \prod_{t=1}^{T} P(y_t|y_{<t}, x) \tag{6}$$

其中 x 是输入 Prompt，y 是生成的推荐理由，T 是序列长度。

4 实现细节

4.1 LLM 生成器模块 (llm_generator.py)

4.1.1 类结构设计

Listing 1: LLMGenerator 类定义

```python
class LLMGenerator:
    def __init__(self, config: ProjectConfig | None = None) -> None:
        self.config = config or ProjectConfig()
        self._client = None
        api_key = self.config.llm_api_key()
        if OpenAI and api_key:
            self._client = OpenAI(api_key=api_key)

    def build_prompt(
        self,
        reference: RecipeRecord,
        candidates: Sequence[RecipeRecord],
        user_input: str,
    ) -> str:
        # Prompt 构建逻辑
        ...

    def generate(
        self,
        reference: RecipeRecord,
        candidates: Sequence[RecipeRecord],
        user_input: str,
    ) -> str:
        # 生成逻辑，含回退策略
        ...
```

4.1.2 Prompt 模板实现

Listing 2: build_prompt 方法实现

```python
def build_prompt(
    self,
    reference: RecipeRecord,
    candidates: Sequence[RecipeRecord],
    user_input: str,
```

```
) -> str:
    parts = [
        "你是一名善于解释口味风格的智能厨房助手。",
        f"用户输入: {user_input}",
        "参考菜谱:",
        reference.as_prompt_chunk(),
        "候选菜谱:",
    ]
    parts.extend(recipe.as_prompt_chunk() for recipe in candidates)
    parts.append(
        "请用中文生成推荐理由，突出共同食材或口味，并给出建议。"
    )
    return "\n\n".join(parts)
```

Prompt 设计要点:

1. **角色设定**: 明确模型作为" 智能厨房助手" 的身份
2. **上下文注入**: 通过 `as_prompt_chunk()` 方法提供结构化菜谱信息
3. **任务指令**: 明确要求" 中文"、" 突出共同食材"、" 给出建议"
4. **分隔符**: 使用双换行分隔各部分，提高可读性

4.1.3 生成与回退策略

Listing 3: generate 方法实现

```
def generate(
    self,
    reference: RecipeRecord,
    candidates: Sequence[RecipeRecord],
    user_input: str,
) -> str:
    prompt = self.build_prompt(reference, candidates, user_input)
    if not self._client:
        return self._fallback_reason(reference, candidates)

    response = self._client.responses.create(
        model=self.config.models.llm_model,
        input=prompt,
    )
    return response.output[0].content[0].text

@staticmethod
def _fallback_reason(
    reference: RecipeRecord,
    candidates: Sequence[RecipeRecord]
) -> str:
    candidate_titles = ", ".join(
        recipe.title for recipe in candidates if recipe.title
    )
    if candidate_titles:
```

```
        return (
            f"你提到的 {reference.title} 与 {candidate_titles} "
            "共享典型食材，因此可以尝试这些菜式来保持相似的风味。"
        )
    return (
        f"根据你对 {reference.title} 的偏好，"
        "我们建议继续探索相同食材/口味的菜式，保持熟悉的风味体验。"
    )
```

4.2 语义向量索引模块 (embeddings.py)

4.2.1 索引构建

Listing 4: RecipeEmbeddingIndex 索引构建

```
class RecipeEmbeddingIndex:
    def __init__(self, model_name: str) -> None:
        self.model_name = model_name
        self._model: SentenceTransformer | None = None
        self._records: dict[str, RecipeRecord] = {}
        self._ids: list[str] = []
        self._matrix: np.ndarray | None = None
        self._enabled = SentenceTransformer is not None

    def build(self, records: Sequence[RecipeRecord]) -> None:
        self._records = {r.recipe_id: r for r in records}
        if not self._enabled:
            return
        self._ensure_model()
        if not self._model:
            return

        texts = [record.as_prompt_chunk() for record in records]
        embeddings = self._model.encode(
            texts,
            show_progress_bar=False,
            convert_to_numpy=True,
            normalize_embeddings=True,  # 归一化以使用内积计算余弦相似度
        )
        self._matrix = embeddings
        self._ids = [record.recipe_id for record in records]
```

4.2.2 向量检索

Listing 5: query 方法实现

```
def query(
    self,
    text: str,
    top_k: int = 5,
    exclude: Sequence[str] | None = None
```

```
) -> list[RecipeRecord]:
    if not self._ready():
        return []

    vector = self._encode_text(text)
    if vector is None:
        return []

    exclude_set = set(exclude or [])
    scores = self._matrix @ vector  # 批量计算相似度

    # 排除指定 ID
    for idx, recipe_id in enumerate(self._ids):
        if recipe_id in exclude_set:
            scores[idx] = -1.0

    # Top-K 检索
    top_k = min(top_k, len(self._ids))
    if top_k <= 0:
        return []

    top_indices = np.argpartition(scores, -top_k)[-top_k:]
    top_sorted = top_indices[np.argsort(scores[top_indices])[::-1]]

    results = []
    for idx in top_sorted:
        if scores[idx] <= 0:
            continue
        record = self._records.get(self._ids[idx])
        if record:
            results.append(record)
    return results
```

4.2.3 菜谱相似度检索

Listing 6: find_similar_to_recipe 方法

```
def find_similar_to_recipe(
    self,
    recipe: RecipeRecord,
    top_k: int = 5
) -> list[RecipeRecord]:
    query_text = recipe.as_prompt_chunk()
    return self.query(
        query_text,
        top_k=top_k + 2,  # 多取几个以补偿排除自身
        exclude=[recipe.recipe_id]
    )
```

4.3 模型加载与异常处理

Listing 7: 模型懒加载与异常处理

```python
def _ensure_model(self) -> None:
    if self._model or not self._enabled:
        return
    try:
        self._model = SentenceTransformer(self.model_name)
    except Exception as exc:
        LOGGER.warning(
            "加载 SentenceTransformer(%s) 失败：%s",
            self.model_name, exc
        )
        self._enabled = False

def _encode_text(self, text: str) -> np.ndarray | None:
    if not self._model:
        return None
    try:
        vector = self._model.encode(
            text,
            show_progress_bar=False,
            convert_to_numpy=True,
            normalize_embeddings=True,
        )
    except Exception as exc:
        LOGGER.warning("文本向量化失败：%s", exc)
        return None
    return vector
```

5 实验设置

5.1 模型配置

表 1: 语义向量模型配置

参数	值
模型名称	sentence-transformers/all-MiniLM-L6-v2
向量维度	384
最大序列长度	256 tokens
归一化	是
相似度度量	余弦相似度（内积）

表 2: LLM 配置

参数	值
Provider	OpenAI
模型	gpt-4o-mini
API 调用方式	Responses API
回退策略	模板化文本

5.2 评估方法

5.2.1 语义检索评估

- **召回准确性**：检查语义相近的菜谱是否被检索到
- **排序质量**：验证相似度分数与人工判断的一致性
- **响应时间**：测量批量编码和检索的延迟

5.2.2 LLM 生成评估

- **流畅性**：生成文本的语法和可读性
- **相关性**：推荐理由是否与输入菜谱相关
- **信息量**：是否提及共同食材、口味等有价值信息
- **一致性**：多次生成的风格是否稳定

6 结果与分析

6.1 语义检索结果

表 3: 语义检索测试用例

查询文本	Top-3 检索结果	评价
番茄炒蛋	番茄豆腐汤、蛋炒饭、西红柿蛋花汤	高度相关
酸甜口味	糖醋排骨、糖醋里脊、番茄炒蛋	语义准确
快手早餐	煎蛋、牛奶燕麦、吐司	场景匹配
川菜	鱼香肉丝、麻婆豆腐、回锅肉	标签相关

6.2 LLM 生成示例

输入：用户查询” 番茄炒蛋”，候选菜谱为番茄豆腐汤、蛋炒西红柿饭
LLM 输出：

> 您提到的番茄炒蛋是一道经典的家常菜，以番茄的酸甜和鸡蛋的嫩滑著称。如果您喜欢这种风味，我推荐尝试以下菜式：
>
> 1. **番茄豆腐汤**：同样以番茄为主料，汤品清淡爽口，豆腐的软嫩与番茄的酸甜形成完美搭配，适合作为配餐汤品。

2. **蛋炒西红柿饭**：将番茄炒蛋的精髓融入主食，鸡蛋与番茄的组合配上米饭，既营养均衡又美味可口。

这些菜式都保留了番茄和鸡蛋的核心风味，同时提供了不同的用餐体验。

6.3 回退策略效果

当 API Key 未配置时，系统生成的模板化理由：

> 你提到的番茄炒蛋与番茄豆腐汤、蛋炒西红柿饭共享典型食材，因此可以尝试这些菜式来保持相似的风味。

虽然信息量较少，但保证了系统的可用性。

6.4 性能指标

表 4: 性能测试结果

操作	平均耗时	说明
模型加载	5-8 秒	首次加载，后续复用
批量编码（约 300 条）	2-3 秒	GPU 可加速
单次查询编码	10-30 ms	
Top-K 检索	< 5 ms	矩阵乘法 + argpartition
LLM 生成	1-3 秒	取决于网络和模型

7 可复现性与代码结构

7.1 相关文件

- `src/graph_rag_recipes/llm_generator.py`：LLM 生成器
- `src/graph_rag_recipes/embeddings.py`：语义向量索引
- `.env.example`：环境变量模板

7.2 环境配置

Listing 8: LLM 相关环境配置

```
# .env 文件配置
OPENAI_API_KEY=sk-your-api-key-here

# 或使用其他 Provider
# OLLAMA_API_KEY=your-ollama-key
# ZHIPU_API_KEY=your-zhipu-key
```

7.3 测试命令

Listing 9: 测试语义检索和 LLM 生成

```
# 安装依赖
uv sync

# 测试向量检索（无需 API Key）
uv run python -c "
from graph_rag_recipes.embeddings import RecipeEmbeddingIndex
from graph_rag_recipes.data_ingest import HowToCookIngestor

ingestor = HowToCookIngestor()
records = ingestor.load_sample_records()
index = RecipeEmbeddingIndex('sentence-transformers/all-MiniLM-L6-v2')
index.build(records)
results = index.query('番茄炒蛋', top_k=3)
for r in results:
    print(r.title)
"

# 测试完整流程（需要 API Key）
uv run scripts/run_pipeline.py "番茄炒蛋"
```

8 总结与未来工作

8.1 工作总结

本人在项目中负责 LLM 与语义生成模块，主要完成了：

1. 设计并实现了结构化的 Prompt 模板，包含角色设定、上下文注入、任务指令
2. 基于 Sentence-Transformers 构建了高效的语义向量索引，支持 O(n) 复杂度的 Top-K 检索
3. 实现了完整的 LLM 调用封装，包含异常处理和优雅降级
4. 为系统的”语义理解”能力提供了核心支撑

8.2 学习收获

- 深入理解了 Prompt Engineering 的设计原则
- 掌握了 Sentence Embeddings 的原理和应用
- 学习了如何设计健壮的 API 封装和降级策略
- 理解了 RAG 系统中”检索”与”生成”的协同关系

8.3 未来改进方向

1. **多 Provider 支持**：完善 Ollama、GLM 等本地/国产模型的适配
2. **Prompt 优化**：引入 Few-shot 示例，提升生成一致性

3. **向量索引优化**：
    - 使用 FAISS 或 Annoy 加速大规模检索
    - 支持增量索引更新
    - 添加索引持久化
4. **生成质量评估**：引入自动化评估指标（BLEU、ROUGE 等）
5. **流式生成**：支持 Streaming 输出提升用户体验
6. **多模态扩展**：引入菜谱图片的视觉特征

参考文献

[1] Reimers, N., & Gurevych, I. (2019). Sentence-BERT: Sentence embeddings using siamese BERT-networks. *EMNLP*.

[2] OpenAI. (2024). OpenAI API Documentation. `https://platform.openai.com/docs`

[3] Wei, J., et al. (2022). Chain-of-thought prompting elicits reasoning in large language models. *NeurIPS*.

[4] Lewis, P., et al. (2020). Retrieval-augmented generation for knowledge-intensive NLP tasks. *NeurIPS*.

[5] Wang, W., et al. (2020). MiniLM: Deep self-attention distillation for task-agnostic compression of pre-trained transformers. *NeurIPS*.

[6] Harris, C. R., et al. (2020). Array programming with NumPy. *Nature*.