生 产 实 习 报 告

生产实习单位 和鲸社区

实 习 时 间 2025 年 7 月 13 日 至

2025 年 8 月 12 日 止

指导人员姓名 吴坚

指导教师姓名 郭玉臣

学 号 2251764

学 生 姓 名 江来

计算机学院（系） 大数据 专业 三 年级

说　明

1、生产实习结束之前，每个学生都必须认真撰写《生产实习报告》。通过撰写生产实习报告，系统地回顾和总结实习的全过程，将实践性教学的感性认知升华到一定的理论高度，从而提高实习教学效果。

2、实习报告要求条理清晰，内容详尽，数据准确。字数一般不少于5000字。

3、实习报告的撰写应符合实习大纲和实习指导书的要求。报告内容可包括：实习日程安排，实习单位情况，专题报告记录的整理，主要设备、工艺流程，技术参数的记录和分析，专题分析，实习收获和体会，合理化建议等。还应附上必要的图纸或表格。（注意不要堆砌技术文档，导致装订困难）

4、生产实习报告的质量反映了生产实习的质量，它是实习成绩评定的主要依据之一。生产实习报告需经实习指导人员审阅，由实习指导教师负责评分。不交实习报告者不得参加实习成绩评定。

一． 实习内容简介

本次实习是同济大学计算机系的集中实习，我选择了和鲸社区作为实习单位。实习采用线上的方式，目标是完成和鲸社区暑期夏令营活动。实习历时一个月，涵盖 python 基础知识、机器学习、深度学习、自然语言处理（NLP）、模型微调、蒸馏与 RAG 等多个领域，从理论学习到实战操作，层层深入，让我对人工智能技术的应用有了系统且深刻的认识。 以下是对各个任务的介绍：

1. Python 基础

主要包含 Python 语法基础以及 numpy、pandas、matplotlib、seaborn 库，每个部分包含若干闯关题，全部满分通过即可通关。其中 Python 语法基础涉及变量类型、条件语句、循环结构、函数定义等核心内容；numpy 库侧重数组运算、矩阵操作等数值计算能力；pandas 库聚焦数据读取、清洗、筛选、聚合等数据处理技巧；matplotlib 和 seaborn 库则围绕数据可视化展开，包括折线图、柱状图、热力图等多种图表的绘制，为后续数据分析与建模提供基础工具支持。

2. 机器学习

涵盖经典算法原理与实战应用。从逻辑回归入手，通过乳腺癌分类与二手车售价预测案例，理解其在分类与回归任务中的应用逻辑，掌握数据划分、交叉验证、评估指标选择等关键环节。后续学习 XGBoost 模型，在葡萄酒多分类和糖尿病指标预测任务中，实践参数调优与特征重要性分析，体会集成学习的优势。最终通过堪培拉天气数据预测综合实战，整合数据预处理、模型选择与优化流程，理解真实场景中算法落地的完整链路。

3. 深度学习

包括线性代数、概率论、信息论与图模型等基础预备知识。核心内容为神经网络原理与模型实践：学习 MLP 结构，在 MNIST 数据集上对比不同激活函数与优化算法的效果，掌握正则化方法；深入 CNN、RNN、自动编码器等基础模型，通过图像卷积运算、LSTM 单元前向传播、自动编码器重构等闯关题，理解 CNN 的局部特征提取、RNN 的时序依赖捕捉、自动编码器的无监督特征学习机制。最后学习 TensorFlow 与 PyTorch 框架，对比静态图与动态图的特点，通过图像分类任务（综合 CNN 与自动编码器）实践模型搭建、训练与评估的完整流程。

4. NLP

以 transformers 框架与 Bert 模型为核心，覆盖多种自然语言处理任务。首先学习 transformers 安装、tokenizers 使用及 Bert 模型结构解析，理解词嵌入、

上下文编码等核心机制。随后开始系列实战深化应用：文本分类、多标签分类、句子相似性识别、命名实体识别、多项选择、文本生成、文本摘要、文本翻译与问答系统。每个任务均包含数据预处理、模型微调与评估环节，全面掌握 NLP 任务的建模思路。

5. 模型微调

实战任务之一，聚焦 LoRA 技术。目标是完成韩语模型微调任务。过程包括大规模数据处理、LoRA 原理理解、模型设置与训练流程实践。体会微调技术在减少计算资源消耗的同时，适配特定语种任务的核心逻辑。

6. 蒸馏

围绕知识蒸馏理论与实战展开，学习大语言模型推理能力、注意力增强技术、知识路由等核心概念，理解通过教师模型向学生模型迁移知识的原理。实战环节基于科学考试任务搭建蒸馏框架，涉及镜像配置、环境安装、数据处理、知识蒸馏训练器构建与评估流程。

7. RAG

聚焦检索增强生成技术在电商场景的应用，目标是搭建电商 RAG 系统，并对比不同 embedding 模型和向量数据库的区别。核心环节包括数据处理、检索系统构建、生成模块设计，以及整体流程集成。

二． 难点问题分析

在本次和鲸社区暑期夏令营实习中，尽管各领域任务层层递进，但每个阶段都存在不同的难点问题，以下是具体分析：

1. python 基础阶段

难点主要在于库的综合应用。单独掌握 numpy 的数组运算或 pandas 的数据清洗方法相对容易，但面对实际闯关题中需要同时调用多个库的场景时，往往会出现逻辑混乱。例如，用 pandas 处理完缺失值后，需用 matplotlib 绘制分布图表，再结合 numpy 计算特征相关性，此时数据格式的转换、参数传递的衔接常出现疏漏，需要反复调试才能打通全流程。

2． 机器学习阶段

核心难点是模型评估与优化的平衡。逻辑回归任务中，单纯追求高准确率可能导致医疗场景下漏诊风险升高，如何在召回率与精确率之间找到平衡点，需要对业务场景有深刻理解；XGBoost 调参时，树深度与正则化参数 C 的组合会显著影响模型泛化能力，参数调整的“蝴蝶效应”往往难以预判，需要通过大量交叉验证才能找到较优解。此外，堪培拉天气预测的综合实战中，特征衍生的合理性直接决定模型性能，如何从温度、湿度等基础特征中提取“体感指数”等高阶特征，非常考验对数据内在关联的敏感度。

3. 深度学习阶段

深度学习阶段的难点集中在模型原理与框架操作的衔接。多层感知机的激活函数选择中，ReLU 的“死亡神经元”问题与 sigmoid 的梯度消失问题难以两全，需要结合网络深度动态调整；CNN 的卷积操作中，步长与 padding 的设置会直接改变特征图尺寸，实际计算时极容易出现维度不匹配问题，非常烦人，得反复推导公式验证；TensorFlow 与 PyTorch 框架的语法差异也带来困扰，例如静态图中变量的生命周期管理与动态图中的即时调试逻辑不同，初期切换框架时经常出现代码运行错误。

4． NLP 阶段

NLP 任务的难点体现在文本预处理与模型适配性上。Bert 模型的 tokenizer 对特殊字符（如韩语中的助词、医学术语的括号注释）处理效果不佳，会导致输入序列断裂，需要自定义分词规则才能缓解；文本生成任务中，生成结果的连贯性与相关性难以兼顾，温度参数调小时输出过于机械，调大后又容易偏离主题，且缺乏直观的评估指标量化生成质量；机器翻译任务中，双语语料的对齐精度直接影响翻译效果，如何处理语序差异较大的语言对，是提升翻译流畅度的关键。

5. 模型微调阶段

LoRA 技术的难点在于低秩矩阵与预训练模型的协同。语料库的多模态数据筛选时，如何剔除噪声样本并保留韩语核心语义，需要结合语言学知识设计过滤规则，否则会导致微调方向偏差；训练过程中，低秩矩阵的秩参数选择直接影响模型适配能力，秩值过小时无法捕捉语种特性，过大则失去参数高效性，且梯度下降过程中容易出现局部最优，需多次调整学习率才能稳定收敛。

6. 知识蒸馏阶段

难点在于师生模型的知识迁移效率。科学考试任务中，教师模型的“黑箱”输出难以转化为学生模型可学习的显性知识，损失函数中蒸馏损失与分类损失的权重分配缺乏标准，导致学生模型要么复刻教师错误，要么丧失推理能力。

在医学考试任务中，数据层面的复杂性进一步加剧了迁移难度。“疾病-症状”的多对多关系使标签体系异常复杂，一种疾病可能对应多种症状（如流感可表现为发热、咳嗽、乏力），一种症状也可能关联多种疾病（如头痛可能源于感冒、高血压或颈椎问题），这种网状关联结构让教师模型的知识难以被学生模型拆解学习。更棘手的是专业术语的处理，医学数据中大量存在的特殊格式表述（如“高血压（继发性）”“心肌梗死（STEMI 型）”），会被分词器误拆分为“高血压”“继发性”等独立 token，破坏术语的完整性，导致学生模型无法理解“继发性高血压”与“原发性高血压”的本质区别，在涉及治疗方案选择的题目中完全失去判断依据。

7. RAG 阶段

RAG 任务的难点在于检索与生成的协同优化。电商数据的多源异构性（商品详情的结构化数据与用户评论的非结构化文本）导致向量生成时特征权重失衡，检索结果常出现“答非所问”；提示词设计需要精准引导模型聚焦检索信息，过于宽松会导致模型“臆想”，过于严格又限制表达灵活性，需要反复测试才能找到平衡点；此外，向量数据库的索引优化涉及底层算法，IVF_FLAT 与 HNSW 等索引的性能差异在数据量激增时会被放大，直接影响系统响应速度。

三． 实习体会与收获

历经一个月的和鲸社区暑期夏令营线上实习，从 Python 基础到 RAG 实战，每一步探索都伴随着挑战与成长，这些经历不仅深化了我对人工智能技术的理解，更让我在实践认知与能力塑造上收获颇丰。

在知识层面，我突破了“碎片化学习”的局限，构建起相对完整的技术体系。从前对机器学习算法的认知停留在孤立的公式推导，而通过乳腺癌分类、天气预测等实战，逐渐理解不同模型在数据特性适配中的逻辑——逻辑回归适合线性可分的简单场景，XGBoost 在处理非线性特征交互时更具优势，这种 “算法-数据-场景”的关联认知，远比课本上的理论更鲜活。深度学习领域，从 MLP 的神经元激活到 CNN 的卷积核滑动，从 RNN 的时序记忆到 Transformer 的自注意力机制，我逐渐看清这些模型突破传统机器学习瓶颈的核心：通过多层非线性变换实现特征的自动抽象，而框架工具（TensorFlow/PyTorch）则是将理论转化为代码的桥梁，其语法差异背后是工程实现思路的分野。

能力提升方面，我的“解决问题”的思维模式被重塑。初期面对 Python 库综合应用的混乱，我学会用“模块化思维”拆解任务——先明确数据处理的每个步骤（清洗→转换→可视化），再逐个攻克库之间的接口适配，这种“拆解-整合”的方法后来在处理 LoRA 微调的数据混乱问题时同样奏效。面对模型训练中的异常，不再是盲目调参，而是学会从损失曲线、特征分布等细节中定位根源：发现蒸馏任务中温度参数设反时，通过对比教师+学生模型的输出分布差异找到了关键证据。这种“观察-假设-验证”的科学流程，让我明白实战能力的核心绝不是记住代码，而是建立排查问题的逻辑链条。

更深刻的体会在于我对“技术落地”有了更深入的认知。课本中完美的算法在实际场景中常遇阻碍：LoRA 微调时，优质语料的缺失比参数调优更致命；知识蒸馏中，教师模型的“隐性知识”难以通过简单损失函数传递；RAG 系统里，用户体验的流畅度不仅取决于模型精度，更依赖数据清洗的细致程度。这些经历让我明白，人工智能是 “数据质量+算法设计+工程优化”的系统工程，其中每个环节的短板都会成为落地瓶颈。

此外，我还收获了不少失败的经历。LoRA 微调与知识蒸馏的多次失败，让我摒弃了“技术万能”的幻想，学会敬畏数据、尊重细节。

最后，这次实习让我对 AI 领域的学习路径有了更清晰的规划。从“跟着教程跑通代码”到“自主设计解决方案”，中间隔着大量踩坑、复盘的循环积累。未来我会更注重理论深度与实战广度的平衡。

四． 见解和建议

通过本次和鲸社区暑期夏令营实习，我对线上实习模式及人工智能学习路径有了更具体的认知，结合自身经历，提出以下见解与建议：

1. 从学习资源角度来看，现有教程在理论与实战的衔接上仍有优化空间。目前多数任务的教程更侧重“怎么做”，但对“为什么这么做”（如参数设计的原理）解释不足。例如在 LoRA 微调任务中，教程仅给出低秩矩阵的秩参数设置示例，却未说明不同秩值对模型性能的影响规律，导致初学者只能盲目试错。建议增加原理+实践对照模块，每个关键参数搭配可视化实验（如不同秩值下的损失曲线对比），帮助学习者理解技术背后的逻辑。

2. 任务设计方面，高难度任务的阶梯性可以进一步细化。LoRA 实战、知识蒸馏等任务对基础要求较高，但当前任务与前置知识的衔接不够紧密——例如蒸馏任务直接要求处理医学数据，却未设置简化版科学数据蒸馏作为过渡，导致初学者难以适应。

3. 题型方面，有些部分的题型实在不合适。比如 NLP，明明是重实践的项目，但是所有闯关题都是基础知识的选择题，建议加入一点实战题目。

4. 技术支持层面，线上实习的调试辅助工具可更完善。一方面，面对代码报错时，现有社区问答多为通用问题，针对夏令营特定任务的解决方案较少；另一方面，线上的实时错误提示以及代码提示不多，主流 ide 如 pycharm，vs 等都已经提供了极为丰富和即时的消息提示以及代码补全，线上平台即使不做到丰富，但是基础的错误提示和代码提示应该是要有的。

对学习者而言，建议在参与实习前做好基础铺垫。可提前掌握 Python 库的综合应用、机器学习基础算法原理，避免在入门阶段因工具使用问题阻碍进度；同时培养文档阅读习惯，遇到问题先查阅官方文档（如 Hugging Face 的 transformers 手册、向量数据库的 API 说明），再结合社区资源解决，这种自主学习能力在技术迭代快速的 AI 领域尤为重要。

所以，本次实习的任务体系覆盖全面，但若能在阶梯性、支持性、资源衔接上进一步优化，可帮助学习者更高效地掌握技术要点，同时减少不必要的挫败感，让线上实习的价值得到更充分的发挥。

五.　附录：三个实战项目的编程文件

1. LoRA

基础作业：

```
!pip install nltk rouge_chinese jieba peft transformers accelerate bitsandbytes -q
#你的代码过程

from nltk.translate.bleu_score import sentence_bleu, SmoothingFunction
from rouge_chinese import Rouge
import jieba
import numpy as np

class TextEvaluator:
    def __init__(self):
        """初始化评估器"""
        self.rouge = Rouge()
        self.smooth = SmoothingFunction()

    def calculate_bleu(self, reference, candidate):
        """
        计算 BLEU 分数

        参数:
            reference (str): 参考文本
            candidate (str): 候选文本

        返回:
            float: BLEU 分数
        """
        # 将文本分词
        reference_tokens = list(jieba.cut(reference))
        candidate_tokens = list(jieba.cut(candidate))

        # 计算 BLEU 分数
        try:
            score = sentence_bleu([reference_tokens], candidate_tokens,
                                  smoothing_function=self.smooth.method1)
            return score
        except Exception as e:
            print(f"计算 BLEU 分数时出错: {e}")
            return 0.0
```

```
39.
40.     def calculate_rouge(self, reference, candidate):
41.         """
42.         计算 ROUGE 分数
43.
44.         参数:
45.             reference (str): 参考文本
46.             candidate (str): 候选文本
47.
48.         返回:
49.             dict: 包含 ROUGE-1、ROUGE-2 和 ROUGE-L 分数的字典
50.         """
51.         try:
52.             scores = self.rouge.get_scores(candidate, reference)[0]
53.             return {
54.                 'rouge-1': scores['rouge-1']['f'],
55.                 'rouge-2': scores['rouge-2']['f'],
56.                 'rouge-l': scores['rouge-l']['f']
57.             }
58.         except Exception as e:
59.             print(f"计算 ROUGE 分数时出错: {e}")
60.             return {'rouge-1': 0.0, 'rouge-2': 0.0, 'rouge-l': 0.0}
61.
62.     def evaluate_text(self, reference, candidate):
63.         """
64.         综合评估文本质量
65.
66.         参数:
67.             reference (str): 参考文本
68.             candidate (str): 候选文本
69.
70.         返回:
71.             dict: 包含所有评估指标的字典
72.         """
73.         # 计算 BLEU 分数
74.         bleu_score = self.calculate_bleu(reference, candidate)
75.
76.         # 计算 ROUGE 分数
77.         rouge_scores = self.calculate_rouge(reference, candidate)
78.
79.         # 返回所有评估指标
80.         return {
81.             'bleu': bleu_score,
82.             **rouge_scores
```

```
        }

# 测试代码
if __name__ == "__main__":
    # 创建评估器实例
    evaluator = TextEvaluator()

    # 测试文本
    reference = "这是一个测试文本，用于评估文本生成质量。"
    candidate = "这是一个测试文本，用来评估生成文本的质量。"

    # 计算评估指标
    scores = evaluator.evaluate_text(reference, candidate)

    # 打印结果
    print("\n 评估结果:")
    print(f"BLEU 分数: {scores['bleu']:.4f}")
        print(f"ROUGE-1 分数: {scores['rouge-1']:.4f}")
        print(f"ROUGE-2 分数: {scores['rouge-2']:.4f}")
        print(f"ROUGE-L 分数: {scores['rouge-l']:.4f}")
    # 文本评估器实现
    from nltk.translate.bleu_score import sentence_bleu,
SmoothingFunction
    from rouge_chinese import Rouge
    import jieba

    class TextEvaluator:
        def __init__(self):
            self.rouge = Rouge()
            self.smooth = SmoothingFunction()

        def calculate_bleu(self, reference, candidate):
            reference_tokens = list(jieba.cut(reference))
            candidate_tokens = list(jieba.cut(candidate))
            try:
                score = sentence_bleu([reference_tokens],
candidate_tokens,
                                      smoothing_function=self.smooth.me
thod1)
                return score
            except:
                return 0.0

        def calculate_rouge(self, reference, candidate):
```

```
            try:
                # 手动分词，增强效果
                ref_cut = ' '.join(jieba.cut(reference))
                cand_cut = ' '.join(jieba.cut(candidate))
                scores = self.rouge.get_scores(ref_cut, cand_cut)[0]
                return {
                    'rouge-1': scores['rouge-1']['f'],
                    'rouge-2': scores['rouge-2']['f'],
                    'rouge-l': scores['rouge-l']['f']
                }
            except Exception as e:
                print(f"计算 ROUGE 分数时出错: {e}")
                return {'rouge-1': 0.0, 'rouge-2': 0.0, 'rouge-l': 0.0}

        def evaluate_text(self, reference, candidate):
            bleu_score = self.calculate_bleu(reference, candidate)
            rouge_scores = self.calculate_rouge(reference, candidate)
            return {'bleu': bleu_score, **rouge_scores}
    # 模拟不同 LoRA 配置的训练输出，并评估其生成质量
    from peft import LoraConfig

    # 模拟生成文本
    reference_text = "这是一个测试文本，用于评估文本生成质量。"
    generated_outputs = {
        (4, 8, 0.05): "这是一个测试文本，用来评估生成文本的质量。",
        (8, 16, 0.1): "这是一个测试文本，用来评估文本质量。",
        (16, 32, 0.1): "这是用于评估文本生成质量的测试文本。",
        (32, 64, 0.15): "这是一个文本测试，用来评估生成的质量。"
    }

    evaluator = TextEvaluator()

    results = []
    for (r, alpha, dropout), generated in generated_outputs.items():
        scores = evaluator.evaluate_text(reference_text, generated)
        results.append({"r": r, "alpha": alpha, "dropout": dropout,
    **scores})

    import pandas as pd
    pd.DataFrame(results)
```

进阶作业：

```
from typing import List, Dict, Set
```

```python
def tokenize_text(text: str, lang: str = None) -> List[str]:
    """根据不同语言使用相应的分词器

    Args:
        text: 要分词的文本
        lang: 语言代码（如果为 None 则自动检测）

    Returns:
        分词后的词语列表
    """
    if not text:
        return []

    # 如果没有指定语言，进行语言检测
    if lang is None:
        lang = detect_language(text)

    # 根据语言选择分词方法
    if lang == 'zh':
        # 中文使用 jieba 分词
        return [word for word in jieba.cut(text) if word.strip()]
    elif lang == 'th':
        # 泰语使用 pythainlp 分词
        return [word for word in thai_tokenize(text) if word.strip()]
    elif lang == 'ko':
        # 韩语分词
        if mecab:
            try:
                # 使用 mecab 进行分词，过滤掉助词等
                return [word for word, pos in mecab.pos(text)
                        if word.strip() and not pos.startswith('J')]
            except Exception:
                pass

        # 如果 mecab 不可用或失败，使用基本分词
        words = []
        current_word = ''
        for char in text:
            if '\uAC00' <= char <= '\uD7AF' or char.isalnum():
                current_word += char
            else:
                if current_word:
                    words.append(current_word)
                current_word = ''
```

```
                if not char.isspace():
                    words.append(char)
        if current_word:
            words.append(current_word)
        return [w for w in words if w.strip()]

    elif lang == 'ru':
        # 俄语分词 - 使用基本的 razdel 分词
        return [token.text for token in ru_tokenize(text)
                if token.text.strip() and not all(c in '.,!?;:()[]{}' for c
in token.text)]

    if lang == 'ar':
        # 标准化阿拉伯文本中的 hamza 字符(ئ/ؤ/إ/أ/ء)为统一形式，原因是阿拉伯语的
变音符号会影响分词结果
        text = araby.normalize_hamza(text)
        # 标准化 alef 字符(ٱ/إ/أ/آ)为基本的 alef(ا)，原因是阿拉伯语的 alef 字符会影
响分词结果
        text = araby.normalize_alef(text)
        # 使用 araby 库对阿拉伯文本进行分词
        words = araby.tokenize(text)
        # 过滤分词结果:
        # 1. 使用 strip()去除首尾空白字符
        # 2. 排除只包含标点符号的词
        # 3. 返回过滤后的词列表
        return [word for word in words if word.strip() and not all(c in
'.,!?;:()[]{}' for c in word)]
    else:
        # 其他语言使用空格分词
        return [word for word in text.split() if word.strip()]

def extract_domain_terms(domain_texts: List[str], general_texts: List[str]
= None, top_n: int = 2000) -> List[str]:
    """使用改进的多语言分词方法提取领域术语"""
    print("开始提取领域术语...")

    # 按语言分组处理文本
    lang_texts = defaultdict(list)
    for text in domain_texts:
        lang = detect_language(text)
        if lang != 'unknown':
            lang_texts[lang].append(text)

    print("\n 文本语言分布:")
```

```
    for lang, texts in lang_texts.items():
        print(f"- {lang}: {len(texts)} 条")

    # 分语言处理并提取术语
    all_terms = []
    for lang, texts in lang_texts.items():
        print(f"\n 处理{lang}语言文本...")

        # 使用语言专属工具提取术语
        lang_terms = set()
        for text in tqdm(texts, desc=f"处理{lang}语言文本"):
            terms = extract_domain_terms_by_language(text, lang)
            lang_terms.update(terms)

        # 将术语集合转换为列表,方便后续排序操作
        lang_terms = list(lang_terms)

        # 计算每个术语在文本中出现的频率
        # 1. 遍历每个文本
        # 2. 对每个文本重新提取术语
        # 3. 使用 Counter 统计所有术语的频率
        term_freq = Counter(t for text in texts for t in
extract_domain_terms_by_language(text, lang))

        # 对术语列表进行排序:
        # 1. 首要排序依据是术语出现频率(term_freq[x])
        # 2. 次要排序依据是术语长度(len(x))
        # reverse=True 表示按降序排列,即频率高的和长度长的排在前面
        lang_terms.sort(key=lambda x: (term_freq[x], len(x)),
reverse=True)

        # 根据语言数量平均分配术语数量配额
        # 1. top_n 是总的期望术语数量
        # 2. len(lang_texts)是语言种类数
        # 3. 对每种语言,只取配额内的高频长术语
        # //是整除运算符,用于计算每种语言分配的术语数量配额
        # 例如:如果 top_n=2000,有 4 种语言,则每种语言分配 2000//4=500 个术语
        # 这里的:是切片操作符,表示从列表开头取到指定位置
        # //是整除运算符,例如 10//3=3
        # 所以 lang_terms[:top_n // len(lang_texts)]表示:
        # 1. 先计算 top_n 除以语言数量的整除结果 n
        # 2. 然后从 lang_terms 列表中取前 n 个元素
        selected_terms = lang_terms[:top_n // len(lang_texts)]
        all_terms.extend(selected_terms)
```

```
        print(f"{lang}语言提取了 {len(selected_terms)} 个术语")
        if selected_terms:
            print(f"{lang}语言术语示例:")
            for term in selected_terms[:5]:
                print(f"  - {term}")

    return all_terms



def extract_domain_terms_by_language(text: str, lang: str) -> List[str]:
    """使用语言专属工具提取领域术语"""
    terms = []

    if lang == 'ko':
        try:
            # 使用 KoNLPy 的 Mecab 分析器
            if mecab:
                # 提取名词和专有名词
                pos_tags = mecab.pos(text)
                terms = [word for word, pos in pos_tags
                         if pos.startswith('NN') or pos.startswith('NNP')]
        except Exception as e:
            print(f"韩语处理失败: {e}")

    elif lang == 'ru':
        try:
            # 使用 pymorphy2 进行形态分析
            if morph:
                words = [t.text for t in ru_tokenize(text)]
                for word in words:
                    parsed = morph.parse(word)[0]
                    # 提取名词和专有名词
                    if 'NOUN' in parsed.tag or 'Name' in parsed.tag:
                        terms.append(word)
        except Exception as e:
            print(f"俄语处理失败: {e}")

    elif lang == 'th':
        try:
            # 使用 pythainlp 进行词性标注
            words = thai_tokenize(text)
            pos_tags = pythainlp.tag.pos_tag(words)
```

```
            # 提取名词和专有名词
            terms = [word for word, pos in pos_tags
                    if pos.startswith('N') or pos.startswith('PROPN')]
        except Exception as e:
            print(f"泰语处理失败: {e}")

    elif lang == 'ar':
        try:
            # 使用 pyarabic 进行基本处理
            text = araby.strip_tashkeel(text)  # 移除变音符号
            words = araby.tokenize(text)
            # 提取可能的术语（长度大于 3 的词）
            terms = [w for w in words if len(w) > 3 and not any(c.isdigit()
for c in w)]
        except Exception as e:
            print(f"阿拉伯语处理失败: {e}")

    return terms


# 加载韩语数据集
file_path =
"/home/mw/input/Russia39613961/part-677f75d865d8-000260.jsonl.gz"
korean_texts = []

with gzip.open(file_path, 'rt', encoding='utf-8') as f:
    for line in f:
        obj = json.loads(line)
        if 'input' in obj and 'output' in obj:
            korean_texts.append(obj['input'] + ' ' + obj['output'])

print(f"已加载韩语样本数量: {len(korean_texts)}")
# 模拟生成结果并评估
class TextEvaluator:
    def __init__(self):
        self.rouge = Rouge()
        self.smooth = SmoothingFunction()

    def calculate_bleu(self, reference, candidate):
        ref_tokens = tokenize_text(reference, 'ko')
        cand_tokens = tokenize_text(candidate, 'ko')
        return sentence_bleu([ref_tokens], cand_tokens,
smoothing_function=self.smooth.method1)
```

```
    def calculate_rouge(self, reference, candidate):
        ref_cut = ' '.join(tokenize_text(reference, 'ko'))
        cand_cut = ' '.join(tokenize_text(candidate, 'ko'))
        scores = self.rouge.get_scores(hyps=cand_cut, refs=ref_cut)
        return scores[0]

    def evaluate_text(self, reference, candidate):
        bleu = self.calculate_bleu(reference, candidate)
        rouge = self.calculate_rouge(reference, candidate)
        return {
            "bleu": bleu,
            "rouge-1": rouge['rouge-1']['f'],
            "rouge-2": rouge['rouge-2']['f'],
            "rouge-l": rouge['rouge-l']['f']
        }
# 模拟 LoRA 配置与生成
reference = "오늘 날씨가 참 맑고 따뜻하네요."
generated_outputs = {
    (4, 8, 0.05): "오늘 날씨가 맑고 따뜻합니다.",
    (8, 16, 0.10): "날씨가 참 맑고 기분이 좋네요.",
    (16, 32, 0.10): "오늘은 따뜻하고 화창한 날씨입니다.",
    (32, 64, 0.15): "기온이 상승하고 있습니다."
}

evaluator = TextEvaluator()
results = []

for (r, alpha, dropout), candidate in generated_outputs.items():
    scores = evaluator.evaluate_text(reference, candidate)
    results.append({"r": r, "alpha": alpha, "dropout": dropout, **scores})

pd.DataFrame(results)
```

2. 知识蒸馏

基础作业：

```
#  计算 BLEU 和 ROUGE

from datasets import load_metric
import numpy as np

class KnowledgeDistillationTrainer:
```

```
    def __init__(self, *args, tokenizer=None, **kwargs):
        ...
        self.tokenizer = tokenizer
        self.bleu = load_metric("bleu")
        self.rouge = load_metric("rouge")
        self.history.update({
            'val_bleu': [],
            'val_rouge': []
        })

    def _evaluate(self, dataloader):
        self.student_model.eval()
        total_loss = 0
        all_preds = []
        all_labels = []
        all_teacher_texts = []
        all_student_texts = []

        with torch.no_grad():
            for batch in tqdm(dataloader, desc="Evaluating"):
                input_ids = batch['input_ids'].to(self.device)
                attention_mask = batch['attention_mask'].to(self.device)
                labels = batch['labels'].to(self.device)

                # 教师模型生成解释
                with torch.no_grad():
                    teacher_ids = self.teacher_model.base_model.generate(
                        input_ids=input_ids,
                        attention_mask=attention_mask,
                        max_length=50,
                        num_beams=4,
                        pad_token_id=self.tokenizer.pad_token_id
                    )
                    teacher_texts =
self.tokenizer.batch_decode(teacher_ids, skip_special_tokens=True)
                    all_teacher_texts.extend(teacher_texts)

                # 学生模型生成解释
                student_ids = self.student_model.base_model.generate(
                    input_ids=input_ids,
                    attention_mask=attention_mask,
                    max_length=50,
                    num_beams=4,
                    pad_token_id=self.tokenizer.pad_token_id
```

```
                )
                student_texts = self.tokenizer.batch_decode(student_ids,
skip_special_tokens=True)
                all_student_texts.extend(student_texts)

                # 原始分类任务评估
                outputs = self.student_model(
                    input_ids=input_ids,
                    attention_mask=attention_mask,
                    labels=labels
                )
                loss = outputs['loss']
                logits = outputs['logits']
                total_loss += loss.item()

                preds = torch.argmax(logits, dim=1).cpu().numpy()
                all_preds.extend(preds)
                all_labels.extend(labels.cpu().numpy())

        # 计算 BLEU 和 ROUGE
        bleu_scores = []
        rouge_scores = []
        for student_text, teacher_text in zip(all_student_texts,
all_teacher_texts):
            # BLEU 需要 reference 是 list of list
            bleu_score = self.bleu.compute(
                predictions=[student_text],
                references=[[teacher_text]]
            )
            bleu_scores.append(bleu_score['bleu'])

            # ROUGE-L
            rouge_score = self.rouge.compute(
                predictions=[student_text],
                references=[teacher_text]
            )
            rouge_scores.append(rouge_score['rougeL'].fmeasure)

        avg_bleu = np.mean(bleu_scores)
        avg_rouge = np.mean(rouge_scores)
        accuracy = accuracy_score(all_labels, all_preds)

        # 记录指标
        self.history['val_bleu'].append(avg_bleu)
```

```
        self.history['val_rouge'].append(avg_rouge)

        return total_loss / len(dataloader), accuracy
#  test1

from datasets import load_metric
import numpy as np
from torch.nn import functional as F

class KnowledgeDistillationTrainer:
    def __init__(self, *args, tokenizer=None, distill_loss_type='kl',
**kwargs):
        ...
        self.tokenizer = tokenizer
        self.bleu = load_metric("bleu")
        self.rouge = load_metric("rouge")
        self.distill_loss_type = distill_loss_type  # 'kl', 'mse', 'bleu',
'rouge'
        self.history.update({
            'val_bleu': [],
            'val_rouge': [],
            'distill_loss': []
        })

    def compute_distill_loss(self, student_logits, teacher_logits,
student_texts=None, teacher_texts=None):
        """计算不同种类的蒸馏损失"""
        if self.distill_loss_type == 'kl':
            # KL 散度损失
            return F.kl_div(
                F.log_softmax(student_logits, dim=-1),
                F.softmax(teacher_logits, dim=-1),
                reduction='batchmean'
            )
        elif self.distill_loss_type == 'mse':
            # MSE 损失
            return F.mse_loss(student_logits, teacher_logits)
        elif self.distill_loss_type == 'bleu' and student_texts is not None:
            # 基于 BLEU 的损失
            bleu_scores = []
            for s_text, t_text in zip(student_texts, teacher_texts):
                score = self.bleu.compute(
                    predictions=[s_text],
                    references=[[t_text]]
```

```
                )['bleu']
                bleu_scores.append(score)
            return 1 - torch.mean(torch.tensor(bleu_scores,
device=student_logits.device))
        elif self.distill_loss_type == 'rouge' and student_texts is not
None:
            # 基于 ROUGE 的损失
            rouge_scores = []
            for s_text, t_text in zip(student_texts, teacher_texts):
                score = self.rouge.compute(
                    predictions=[s_text],
                    references=[t_text]
                )['rougeL'].fmeasure
                rouge_scores.append(score)
            return 1 - torch.mean(torch.tensor(rouge_scores,
device=student_logits.device))
        else:
            raise ValueError(f"Unknown distill loss type:
{self.distill_loss_type}")

    def _evaluate(self, dataloader):
        self.student_model.eval()
        total_loss = 0
        total_distill_loss = 0
        all_preds = []
        all_labels = []
        all_teacher_texts = []
        all_student_texts = []

        with torch.no_grad():
            for batch in tqdm(dataloader, desc="Evaluating"):
                input_ids = batch['input_ids'].to(self.device)
                attention_mask = batch['attention_mask'].to(self.device)
                labels = batch['labels'].to(self.device)

                # 教师模型生成解释和 logits
                with torch.no_grad():
                    teacher_outputs = self.teacher_model(
                        input_ids=input_ids,
                        attention_mask=attention_mask,
                        labels=labels,
                        output_hidden_states=True
                    )
                    teacher_logits = teacher_outputs['logits']
```

```
                teacher_ids = self.teacher_model.base_model.generate(
                    input_ids=input_ids,
                    attention_mask=attention_mask,
                    max_length=50,
                    num_beams=4,
                    pad_token_id=self.tokenizer.pad_token_id
                )
                teacher_texts =
self.tokenizer.batch_decode(teacher_ids, skip_special_tokens=True)
                all_teacher_texts.extend(teacher_texts)

            # 学生模型生成解释和 logits
            student_outputs = self.student_model(
                input_ids=input_ids,
                attention_mask=attention_mask,
                labels=labels,
                output_hidden_states=True
            )
            student_logits = student_outputs['logits']
            loss = student_outputs['loss']

            student_ids = self.student_model.base_model.generate(
                input_ids=input_ids,
                attention_mask=attention_mask,
                max_length=50,
                num_beams=4,
                pad_token_id=self.tokenizer.pad_token_id
            )
            student_texts = self.tokenizer.batch_decode(student_ids,
skip_special_tokens=True)
            all_student_texts.extend(student_texts)

            # 计算蒸馏损失
            distill_loss = self.compute_distill_loss(
                student_logits,
                teacher_logits,
                student_texts if self.distill_loss_type in ['bleu',
'rouge'] else None,
                teacher_texts if self.distill_loss_type in ['bleu',
'rouge'] else None
            )

            total_loss += loss.item()
```

```
                total_distill_loss += distill_loss.item()
                preds = torch.argmax(student_logits, dim=1).cpu().numpy()
                all_preds.extend(preds)
                all_labels.extend(labels.cpu().numpy())

        # 计算文本生成指标
        bleu_scores = []
        rouge_scores = []
        for student_text, teacher_text in zip(all_student_texts,
all_teacher_texts):
            # BLEU 需要 reference 是 list of list
            bleu_score = self.bleu.compute(
                predictions=[student_text],
                references=[[teacher_text]]
            )
            bleu_scores.append(bleu_score['bleu'])

            # ROUGE-L
            rouge_score = self.rouge.compute(
                predictions=[student_text],
                references=[teacher_text]
            )
            rouge_scores.append(rouge_score['rougeL'].fmeasure)

        avg_bleu = np.mean(bleu_scores)
        avg_rouge = np.mean(rouge_scores)
        accuracy = accuracy_score(all_labels, all_preds)
        avg_distill_loss = total_distill_loss / len(dataloader)

        # 记录指标
        self.history['val_bleu'].append(avg_bleu)
        self.history['val_rouge'].append(avg_rouge)
        self.history['distill_loss'].append(avg_distill_loss)

        return total_loss / len(dataloader), accuracy
```

进阶作业：

```
def process_for_training(self, tokenizer, max_length=512):
    """处理数据为训练格式，适配新数据格式"""
    if len(self.data) == 0:
        raise ValueError("没有可处理的数据，请先确保数据已正确加载")

    processed_data = {'input_ids': [], 'attention_mask': [], 'labels': []}
```

```
    logger.info(f"处理 {len(self.data)} 条数据记录...")
    for idx, item in enumerate(self.data):
        try:
            # 验证数据格式
            if 'question' not in item:
                logger.warning(f"数据项 {idx} 缺少 'question' 字段，跳过")
                continue
            if 'options' not in item or not item['options']:
                logger.warning(f"数据项 {idx} 缺少选项，跳过")
                continue
            if 'answer_idx' not in item:
                logger.warning(f"数据项 {idx} 缺少答案标记，跳过")
                continue

            question = item['question']
            options = item['options']
            answer_idx = item['answer_idx']

            # 验证答案格式是否为 A/B/C/D
            if answer_idx not in ['A', 'B', 'C', 'D']:
                logger.warning(f"数据项 {idx} 答案格式不正确: {answer_idx}，
跳过")
                continue

            correct_idx = ord(answer_idx) - ord('A')  # 转换为 0~3

            # 为每个选项生成一个样本
            for i, (label, text) in enumerate(options.items()):
                # 构造输入文本
                text_input = f"问题：{question}\n 选项：{text}\n 这个选项是否
正确？"

                # 编码文本
                encoding = tokenizer(text_input, max_length=max_length,
padding='max_length', truncation=True)

                # 添加到处理后的数据中
                processed_data['input_ids'].append(encoding['input_ids'])
                processed_data['attention_mask'].append(encoding['attenti
on_mask'])

                # 判断是否为正确选项
                is_correct = 1 if i == correct_idx else 0
                processed_data['labels'].append(is_correct)
```

```
        except Exception as e:
            logger.warning(f"处理数据项 {idx} 时出错: {str(e)}")
            continue

    # 检查是否有有效处理的数据
    if not processed_data['input_ids']:
        raise ValueError("处理后没有有效的训练数据")

    # 转换为 PyTorch 张量
    for key in processed_data:
        processed_data[key] = torch.tensor(processed_data[key])

    logger.info(f"数据处理完成，共生成 {len(processed_data['input_ids'])} 个训练样本")
    return processed_data
```

3. RAG

基础作业：

```
'''这个 cell 中加载不同的 Embedding 模型'''
## 关键代码
# Requires transformers>=4.51.0
# Requires sentence-transformers>=2.7.0

from sentence_transformers import SentenceTransformer

# Load the model
model1 =
SentenceTransformer("/home/mw/.cache/modelscope/hub/models/Qwen/Qwen3-Embedding-0.6B")
model2 =
SentenceTransformer("/home/mw/.cache/modelscope/hub/models/Qwen/Qwen3-Embedding-4B")
#model3 =
SentenceTransformer("/home/mw/.cache/modelscope/hub/models/Qwen/Qwen3-Embedding-8B")
# 8B 不能用，cuda memory 不足

documents = [
    "The capital of China is Beijing.",
```

```
    "Gravity is a force that attracts two bodies towards each other. It gives
weight to physical objects and is responsible for the movement of planets
around the sun.",
]

document_embeddings1 = model1.encode(documents)
document_embeddings2 = model2.encode(documents)
#document_embeddings3 = model3.encode(documents)

'''粘贴教程的代码'''
import os
import sys
import torch
import random
import numpy as np
import logging
from pathlib import Path
from collections import Counter
from datetime import datetime
from sklearn.metrics.pairwise import cosine_similarity
from sklearn.feature_extraction.text import TfidfVectorizer
from sentence_transformers import SentenceTransformer
from transformers import AutoModelForCausalLM, AutoTokenizer
import argparse
import time # Added for performance monitoring
import re  # For text processing

# Try importing NLTK components with error handling
try:
    from nltk.translate.bleu_score import sentence_bleu,
SmoothingFunction  # For BLEU score calculation
    NLTK_AVAILABLE = True
except ImportError:
    logging.warning("NLTK 库导入失败，BLEU 分数评估将被禁用")
    NLTK_AVAILABLE = False

# ========== 身份声明自动应答机制 ==========
IDENTITY_ANSWER = "您好，我是客服小助手，你问的是：\""
IDENTITY_QUESTIONS = [
    "你是什么模型", "你是谁", "你是谁的问题", "你是什么模型相关的问题", "你是
什么模型相关的问题", "你是谁的问题", "你是谁", "你是什么模型"
]

def check_identity_question(query):
```

```
    for q in IDENTITY_QUESTIONS:
        if q in query:
            return True
    return False

# ========== 日志设置 ==========
logging.basicConfig(level=logging.INFO,
format='%(asctime)s %(levelname)s %(message)s')
'''粘贴教程的代码'''
# ========== 数据收集与预处理 ==========
# 电商知识文档
RAW_DOCS = [
    # 商品信息
    "商品 A：高性能笔记本电脑，16GB 内存，512GB SSD，适合办公与游戏，售价 5999 元，支持分期付款。",
    "商品 B：无线蓝牙耳机，降噪功能，续航 30 小时，适合运动与通勤，售价 499 元，赠送收纳盒。",
    "商品 C：智能手表，支持心率监测、睡眠分析、运动追踪，防水 50 米，续航 7 天，售价 1299 元。",
    "商品 D：家用智能扫地机器人，激光导航，APP 控制，自动回充，适合各种地板清洁，售价 2499 元。",
    "商品 E：专业级数码相机，2400 万像素，4K 视频拍摄，防抖功能，含 18-55mm 标准镜头，售价 6299 元。",
    "商品 F：便携式蓝牙音箱，360°环绕立体声，防水防尘，续航 12 小时，支持 TWS 双音箱连接，售价 299 元。",
    "商品 G：多功能料理机，搅拌、切碎、榨汁多合一，2000W 大功率，8 档调速，静音设计，售价 899 元。",

    # 促销政策
    "促销政策：满 1000 减 100，部分商品参与，详情请咨询客服。",
    "限时特惠：每日 10 点、14 点、20 点开启秒杀，低至 5 折，每人限购 1 件。",
    "会员专享：银卡会员 95 折，金卡会员 9 折，钻石会员 85 折，不与其他优惠同享。",
    "新人福利：首次下单立减 50 元，无门槛使用，有效期 7 天。",
    "节日活动：618 年中大促，全场商品满减，部分商品买二送一。",
    "积分兑换：消费 1 元积 1 分，积分可兑换优惠券或实物礼品，详见积分商城。",
    "优惠券规则：优惠券不可叠加使用，不可与满减活动同享，有效期请见券面说明。",

    # 售后服务
    "售后服务：7 天无理由退换货，1 年质保，支持全国联保。",
    "退货政策：商品签收后 7 天内可申请无理由退货，商品需保持原包装及完好，退回运费由买家承担。",
    "换货流程：联系客服提交换货申请，审核通过后寄回商品，收到退回商品后 3 个工作日内发出新商品。",
```

```
    "保修条款：电子产品享受 1 年免费保修，人为损坏、擅自拆机、进水或改装不在保修范围内。",
    "售后网点：全国设有 1000+售后服务网点，可提供上门维修或到店维修服务。",
    "延长保修：可购买延长保修服务，最多可延长至 3 年，费用为商品价格的 5%-10%。",

    # 物流配送
    "物流说明：订单 24 小时内发货，支持多家快递，包邮服务。",
    "配送范围：全国大部分地区支持配送，港澳台及偏远地区可能产生额外运费。",
    "运费规则：单笔订单满 99 元免运费，不满 99 元收取 10 元运费，特大件商品另计。",
    "发货时间：工作日 16 点前下单当天发货，节假日及特殊情况可能顺延。",
    "自提服务：支持就近门店自提，下单时选择自提点，收到提货通知后凭码取货。",
    "极速达：部分城市支持 2 小时极速达服务，订单满足条件可在下单页面选择。",

    # 支付方式
    "支付方式：支持支付宝、微信支付、银联、信用卡等多种支付方式。",
    "分期付款：单笔订单满 500 元可申请 3-24 期分期，部分银行卡用户可享免息特权。",
    "货到付款：特定区域支持货到付款服务，需支付 5 元手续费。",
    "发票开具：可开具电子发票或纸质发票，请在下单时选择，纸质发票将随商品一起寄出。",

    # 会员体系
    "会员等级：普通会员、银卡会员（累计消费 5000 元）、金卡会员（累计消费 20000 元）、钻石会员（累计消费 50000 元）。",
    "会员权益：专属客服、生日礼遇、提前购、专享折扣、积分加速、免费试用等，等级越高权益越多。",
    "积分规则：消费 1 元获得 1 积分，参与活动可获得额外积分，积分有效期为一年。",

    # 常见问题
    "订单修改：订单支付成功后，如需修改收货信息请立即联系客服，发货后无法修改。",
    "账户安全：定期修改密码，不要在不信任的设备上登录账号，警惕钓鱼网站和诈骗信息。",
    "商品咨询：关注商品参数、适用场景等问题可咨询在线客服或拨打服务热线 400-888-8888。",
    "投诉建议：对服务不满意可通过 APP 意见反馈或发送邮件至 service@example.com 进行投诉。"
]

# ========== 文档分块 ==========
def chunk_docs(docs, chunk_size=50, overlap=10):
    """
```

```
    将输入的文档列表按指定的字符数进行分块（chunk），支持分块之间的重叠。

    参数说明：
        docs (List[str]): 输入的文档列表，每个元素为一个字符串，代表一篇文档。
        chunk_size (int): 每个分块的最大字符数。默认值为 50。
        overlap (int): 相邻分块之间的重叠字符数。默认值为 10。

    实现细节：
        - 对于每一篇文档，从头开始，按照 chunk_size 的长度截取一段文本作为一个分块。
        - 每次分块的起始位置向后移动(chunk_size - overlap)个字符，从而实现分块之间的重叠。
        - 如果分块的终止位置已经到达或超过文档末尾，则最后一个分块会自动截断到文档结尾。
        - 该方法适用于短文本或中等长度文档的简单分块，便于后续向量化检索。

    返回值：
        List[str]: 分块后的所有文本块组成的列表。

    示例：
        输入: ["abcdefg", "hijklmnop"], chunk_size=4, overlap=2
        输出: ['abcd', 'cdef', 'efg', 'hijk', 'ijkl', 'ijkl', 'klmn', 'mnop', 'nop']

    """
    chunks = []
    for doc in docs:
        start = 0
        while start < len(doc):
            end = min(start + chunk_size, len(doc))
            chunk = doc[start:end]
            chunks.append(chunk)
            if end == len(doc):
                break
            start += chunk_size - overlap
    return chunks
'''粘贴教程的代码'''
from transformers import AutoTokenizer, AutoModel
import torch
# ========== 向量化 ==========
def build_tfidf(chunks):
    vectorizer = TfidfVectorizer()
    tfidf_matrix = vectorizer.fit_transform(chunks)
    # print("分词后的单词内容：")
```

```
    # print(vectorizer.get_feature_names_out())

    return vectorizer, tfidf_matrix


#
==========================================================================
=======================
#
==========================================================================
=======================
# 修改这里，返回不同的模型
def build_dense_encoder(num):
    if(num==1):
        print("正在测试 embedding 0.6B")
        return model1
    elif(num==2):
        print("正在测试 embedding 4B")
        return model2
    elif(num==3):
        return "这个 cuda memory 不够，不能用"
#
==========================================================================
=======================
#
==========================================================================
=======================

def encode_dense(dense_encoder, chunks, is_query=False):
    return dense_encoder.encode(chunks, is_query=is_query)

# ========== 内存型向量数据库 ==========
class SimpleVectorDB:
    def __init__(self, chunks, tfidf_matrix, tfidf_vectorizer,
dense_embeds, dense_encoder=None):
        self.chunks = chunks
        self.tfidf_matrix = tfidf_matrix
        self.tfidf_vectorizer = tfidf_vectorizer
        self.dense_embeds = dense_embeds
        self.dense_encoder = dense_encoder

    def search(self, query, topk=3, mode="hybrid", dense_encoder=None):
        results = []
        scores_sparse = None
```

```
        scores_dense = None

        # 处理 query 格式
        if isinstance(query, str):
            query_text = query
        elif isinstance(query, dict) and 'query' in query:
            query_text = query['query']
        else:
            query_text = str(query)

        # 如果未传 dense_encoder，使用对象自身的
        if dense_encoder is None and hasattr(self, 'dense_encoder'):
            dense_encoder = self.dense_encoder

        if mode in ["sparse", "hybrid"]:
            q_vec = self.tfidf_vectorizer.transform([query_text])
            scores_sparse = cosine_similarity(q_vec, self.tfidf_matrix)[0]

        # if mode in ["dense", "hybrid"] and dense_encoder is not None:
        #     # 去掉 show_progress_bar 和 prompt_name
        #     q_dense = dense_encoder.encode([query_text],
is_query=True)[0]
        #     scores_dense = cosine_similarity([q_dense],
self.dense_embeds)[0]

        if mode in ["dense", "hybrid"] and dense_encoder is not None:
            q_dense = dense_encoder.encode([query_text],
show_progress_bar=False, prompt_name="query")[0]
            scores_dense = cosine_similarity([q_dense],
self.dense_embeds)[0]


        # 融合分数
        if mode == "sparse":
            scores = scores_sparse
        elif mode == "dense":
            scores = scores_dense
        else:  # hybrid
            if scores_sparse is not None and scores_dense is not None:
                scores = 0.5 * scores_sparse + 0.5 * scores_dense
            elif scores_sparse is not None:
                scores = scores_sparse
            elif scores_dense is not None:
                scores = scores_dense
```

```
            else:
                return []

        top_idx = np.argsort(scores)[::-1][:topk]
        for idx in top_idx:
            results.append((self.chunks[idx], float(scores[idx])))
        return results

# ========== 生成模块 ==========
def load_generation_model(model_path, device):
    model = AutoModelForCausalLM.from_pretrained(
        model_path,
        trust_remote_code=True,
        torch_dtype=torch.float16,
    ).to(device)
    tokenizer = AutoTokenizer.from_pretrained(
        model_path,
        trust_remote_code=True,
    )
    return model, tokenizer

def generate_answer(model, tokenizer, context, query, device,
max_new_tokens=256):
    prompt = f"""已知信息：{context}
                用户提问：{query}
                请基于已知信息直接回答用户问题。回答需要：
                1. 简明扼要，不要重复已知信息
                2. 如果已知信息中没有相关内容，明确告知用户
                3. 不要包含任何推理过程
                4. 不要在回答中包含"根据已知信息"之类的提示语
                回答："""

    inputs = tokenizer(prompt, return_tensors="pt").to(device)
    with torch.no_grad():
        outputs = model.generate(**inputs, max_new_tokens=max_new_tokens,
do_sample=True, top_p=0.95)
    full_response = tokenizer.decode(outputs[0],
skip_special_tokens=True)

    # Extract only the answer part
    answer = full_response[len(prompt):].strip() if
full_response.startswith(prompt) else full_response

    # Remove any remaining reasoning patterns
```

```
    answer = answer.split("回答：")[-1] if "回答：" in answer else answer

    return answer.strip()

# ========== 系统评估与优化 ==========
def evaluate_retrieval_performance(retrieved_docs, relevant_docs):
    """
    评估检索性能

    参数:
        retrieved_docs (List[str]): 系统检索到的文档
        relevant_docs (List[str]): 人工标注的相关文档

    返回:
        Dict: 包含检索评估指标的字典
    """
    # 计算召回率
    def calculate_recall():
        relevant_retrieved = set(retrieved_docs) & set(relevant_docs)
        return len(relevant_retrieved) / len(relevant_docs) if
relevant_docs else 0.0

    # 计算精确率
    # 精确率（Precision）是指在所有被系统检索出来的文档（retrieved_docs）中，
有多少比例是真正相关的文档（relevant_docs）。
    # 计算方法是：用检索到的相关文档数量（relevant_retrieved）除以总共检索到的
文档数量（retrieved_docs）。
    # 如果没有检索到任何文档，则精确率为 0.0。
    def calculate_precision():
        relevant_retrieved = set(retrieved_docs) & set(relevant_docs)  # 检
索结果与相关文档的交集，即被正确检索出来的相关文档
        return len(relevant_retrieved) / len(retrieved_docs) if
retrieved_docs else 0.0

    # 计算 F1 分数
    def calculate_f1():
        precision = calculate_precision()
        recall = calculate_recall()
        return 2 * (precision * recall) / (precision + recall) if (precision
+ recall) > 0 else 0.0

    # 计算平均排名 (MRR)
    def calculate_mrr():
        if not relevant_docs or not retrieved_docs:
```

```
            return 0.0

        # 找到第一个相关文档的排名
        for i, doc in enumerate(retrieved_docs):
            if doc in relevant_docs:
                return 1.0 / (i + 1)  # 排名从1开始
        return 0.0

    precision = calculate_precision()
    recall = calculate_recall()
    f1 = calculate_f1()
    mrr = calculate_mrr()

    logging.info(f"检索评估 - 精确率: {precision:.2f}, 召回率: {recall:.2f},
F1: {f1:.2f}, MRR: {mrr:.2f}")

    return {
        "precision": precision,
        "recall": recall,
        "f1": f1,
        "mrr": mrr
    }

def evaluate_answer_quality(answer, reference_docs, query):
    """
    评估生成答案的质量

    参数:
        answer (str): 生成的答案文本
        reference_docs (List[str]): 检索到的参考文档列表
        query (str): 用户的原始问题

    返回:
        Dict: 包含各项评估指标的字典
    """
    # 事实准确性评估（适用于中文，基于最长公共子串覆盖率）
    def check_factual_accuracy(answer, docs):
        # 合并所有参考文档为一个字符串
        doc_text = "".join(docs).replace("，", "").replace("。",
"").replace("：", "").replace("！", "").replace("？", "").replace("、",
"").replace("；", "")
        answer_text = answer.replace("，", "").replace("。", "").replace("：
", "").replace("！", "").replace("？", "").replace("、", "").replace("；
", "")
```

```
        # 如果任一为空，返回 0
        if not doc_text or not answer_text:
            return 0.0

        # 以 n-gram 方式（如 2-gram 或 3-gram）统计 answer 中有多少片段在 doc_text
中出现
        def get_ngrams(text, n=2):
            return {text[i:i+n] for i in range(len(text)-n+1)} if
len(text) >= n else set()

        # 可以尝试 2-gram 和 3-gram 的平均
        ngram_matches = []
        ngram_total = 0
        for n in [2, 3]:
            answer_ngrams = get_ngrams(answer_text, n)
            doc_ngrams = get_ngrams(doc_text, n)
            if answer_ngrams:
                match_count = len(answer_ngrams & doc_ngrams)
                ngram_matches.append(match_count / len(answer_ngrams))
                ngram_total += 1

        # 如果没有任何 ngram，返回 0
        if not ngram_matches:
            return 0.0

        # 返回平均覆盖率
        return sum(ngram_matches) / ngram_total

    # 上下文相关性评估
    def check_context_relevance(answer, docs, query):
        # 使用向量相似度计算答案与文档的相关性
        try:
            vectorizer = TfidfVectorizer()
            docs_text = " ".join(docs)
            texts = [answer, docs_text, query]
            if all(text.strip() for text in texts):  # 确保所有文本非空
                vectors = vectorizer.fit_transform(texts)
                relevance_score = cosine_similarity(vectors[0:1],
vectors[1:2])[0][0]
                return relevance_score
            else:
                print("计算相关性为 0，因为所有文本为空")
        except Exception as e:
```

```
            logging.warning(f"计算相关性时出错: {str(e)}")
        return 0.0

    # 幻觉检测
    def check_hallucination(answer, docs):
        """检测生成内容中有多少连字出现在参考文档中的连字"""
        # 连字定义为连续两个字符的子串
        def get_bigrams(text):
            text = text.replace("，", "").replace("。", "").replace("： ", "")
            return [text[i:i+2] for i in range(len(text)-1)] if len(text) >=
2 else []

        # 获取参考文档所有连字集合
        doc_text = " ".join(docs)
        doc_bigrams = set(get_bigrams(doc_text))

        # 获取答案中的连字
        answer_bigrams = get_bigrams(answer)

        if not answer_bigrams:
            return 0.0

        # 计算答案中有多少连字出现在参考文档中
        matched_bigrams = [bg for bg in answer_bigrams if bg in doc_bigrams]
        match_rate = len(matched_bigrams) / len(answer_bigrams)

        return match_rate

    # 计算 BLEU 分数（支持中文，自动按字切分）
    def calculate_bleu(answer, docs):
        """
        计算生成答案与参考文档之间的 BLEU 分数（支持中文，自动按字切分）。

        BLEU（Bilingual Evaluation Understudy）是一种常用的自动化评估指标，
用于衡量生成文本与参考文本之间的相似度，常用于机器翻译和文本生成任务。

        具体实现步骤如下：
        1. 首先将所有参考文档（docs）合并后，按句子进行分割，得到参考句子列表
reference_sentences。
        2. 同样将生成的答案（answer）按句子分割，得到答案句子列表
answer_sentences。
        3. 对于答案中的每一个句子，进行如下操作：
            a. 将该句子分词（这里简单用 split()，假设已分好词）。
            b. 将所有参考句子也分词，作为参考（reference）列表。
```

```
            c. 使用 NLTK 的 sentence_bleu 函数，计算该答案句子与所有参考句子的
BLEU 分数。
            d. 使用 SmoothingFunction().method1 进行平滑处理，避免短句 BLEU
为 0。
        4. 对所有答案句子的 BLEU 分数取平均，作为最终的 BLEU 分数。

        注意事项：
        - BLEU 分数范围为 0~1，越高表示生成内容与参考内容越接近。
        - 这里的实现是“句级 BLEU”，即对每个答案句子分别计算，然后取平均。
        - 参考文档和答案都假设已经分词（如中文可用空格分词）。
        - 若 NLTK 不可用或输入为空，返回 0.0。

        """
        if not NLTK_AVAILABLE:
            return 0.0

        try:
            # 1. 将参考文档分割为句子
            reference_sentences = []
            for doc in docs:
                # 按中英文标点分句
                sentences = re.split(r'[。！？!?.]', doc)
                sentences = [s.strip() for s in sentences if s.strip()]
                reference_sentences.extend(sentences)
            if not reference_sentences:
                return 0.0

            # 2. 将答案分割为句子
            answer_sentences = re.split(r'[。！？!?.]', answer)
            answer_sentences = [s.strip() for s in answer_sentences if
s.strip()]
            if not answer_sentences:
                return 0.0

            # 3. 对每个答案句子计算 BLEU 分数（按字切分）
            bleu_scores = []
            smoothie = SmoothingFunction().method1  # 平滑处理，避免短句 BLEU
为 0
            # 参考句子全部按字切分
            ref_tokens = [list(ref_sent) for ref_sent in reference_sentences
if ref_sent]
            for ans_sent in answer_sentences:
                ans_tokens = list(ans_sent)  # 按字切分
                if ans_tokens:
```

```
                    bleu = sentence_bleu(ref_tokens, ans_tokens,
smoothing_function=smoothie)
                    bleu_scores.append(bleu)
            return np.mean(bleu_scores) if bleu_scores else 0.0

        except Exception as e:
            logging.warning(f"计算 BLEU 分数时出错: {str(e)}")
            return 0.0

    # 计算困惑度 (Perplexity) - 使用简化方法评估流畅度
    def calculate_perplexity(answer):
        """使用简化方法估算困惑度 - 评估生成文本的流畅度
            在实际中，perplexity = exp(-1/N * ∑log P(w_i|context))
            其中 N 是词数，P(w_i|context)是每个词在上下文中的条件概率
            具体实现：
            使用预训练语言模型(如 GPT、BERT)计算每个 token 的概率
            需要整个模型的前向推理过程
            通常基于大规模语料库训练的模型
        """
        try:
            # 简化实现：使用句子长度、标点符号比例等作为流畅度的启发式指标

            # 1. 检查句子长度分布 (过短或过长的句子可能不流畅)
            sentences = re.split(r'[。！？!?.]', answer)
            sentences = [s.strip() for s in sentences if s.strip()]
            if not sentences:
                return 0.0  # 没有完整句子

            sent_lengths = [len(s) for s in sentences]
            avg_length = np.mean(sent_lengths) if sent_lengths else 0

            # 长度异常惩罚：句子过短(<5)或过长(>50)会降低流畅度
            length_penalty = sum(1 for length in sent_lengths if length <
5 or length > 50) / len(sentences)

            # 2. 检查标点符号比例
            punctuation_marks = re.findall(r'[，。！？、：；""''（）【】《》]',
answer)
            punct_ratio = len(punctuation_marks) / len(answer) if answer
else 0

            # 标点过多或过少都会影响流畅度 (理想范围大约是 0.1-0.2)
            punct_penalty = abs(punct_ratio - 0.15) / 0.15
```

```
            # 3. 检查重复词语
            words = answer.split()
            word_counts = Counter(words)
            repetition_ratio = 1 - (len(word_counts) / len(words)) if words
else 0

            # 计算综合流畅度分数 (1 为最流畅，0 为最不流畅)
            fluency_score = max(0, 1 - (length_penalty * 0.3 + punct_penalty
* 0.3 + repetition_ratio * 0.4))

            # 将流畅度转换为困惑度的近似值 (困惑度越低越好)
            # 困惑度范围约定为 1-50，1 代表最流畅
            approx_perplexity = 1 + (1 - fluency_score) * 49

            return approx_perplexity

        except Exception as e:
            logging.warning(f"计算困惑度时出错: {str(e)}")
            return 50.0  # 返回最大困惑度值

    # 返回评估结果
    factual_accuracy = check_factual_accuracy(answer, reference_docs)
    print(f"factual_accuracy: {factual_accuracy}")
    context_relevance = check_context_relevance(answer, reference_docs,
query)
    print(f"context_relevance: {context_relevance}")
    hallucination_rate = check_hallucination(answer, reference_docs)
    print(f"hallucination_rate: {hallucination_rate}")
    # 尝试计算 BLEU 分数
    bleu_score = 0.0
    try:
        bleu_score = calculate_bleu(answer, reference_docs)
    except Exception as e:
        logging.warning(f"BLEU 分数计算失败: {str(e)}")

    # 计算困惑度
    perplexity = 50.0  # 默认值 (越高表示越不流畅)
    try:
        perplexity = calculate_perplexity(answer)
    except Exception as e:
        logging.warning(f"困惑度计算失败: {str(e)}")

    # 输出评估结果到日志
```

```
    logging.info(f"答案质量评估 - 事实准确性: {factual_accuracy:.2f}, 上下文相关性: {context_relevance:.2f}, " +
                 f"幻觉率: {hallucination_rate:.2f}, BLEU: {bleu_score:.4f}, 困惑度: {perplexity:.2f}")

    return {
        "factual_accuracy": factual_accuracy,
        "context_relevance": context_relevance,
        "hallucination_rate": hallucination_rate,
        "bleu_score": bleu_score,
        "perplexity": perplexity
    }

class PerformanceMonitor:
    """性能监控类"""
    def __init__(self):
        self.metrics = {
            "response_times": [],
            "memory_usage": [],
            "gpu_usage": [],
            "error_counts": Counter()
        }

    def record_response_time(self, start_time, end_time):
        """记录响应时间"""
        response_time = end_time - start_time
        self.metrics["response_times"].append(response_time)

    def record_resource_usage(self):
        """记录资源使用情况"""
        if torch.cuda.is_available():
            gpu_memory = torch.cuda.memory_allocated() / 1024**2  # MB
            self.metrics["gpu_usage"].append(gpu_memory)

    def record_error(self, error_type):
        """记录错误"""
        self.metrics["error_counts"][error_type] += 1

    def get_statistics(self):
        """获取统计信息"""
        if not self.metrics["response_times"]:
            return {}

        return {
```

```
            "avg_response_time": np.mean(self.metrics["response_times"]),
            "p95_response_time":
np.percentile(self.metrics["response_times"], 95),
            "avg_gpu_usage": np.mean(self.metrics["gpu_usage"]) if
self.metrics["gpu_usage"] else 0,
            "error_statistics": dict(self.metrics["error_counts"])
        }

class OptimizationManager:
    """优化管理器"""
    def __init__(self, vectordb, model, tokenizer):
        self.vectordb = vectordb
        self.model = model
        self.tokenizer = tokenizer
        self.performance_monitor = PerformanceMonitor()

    def optimize_retrieval_weights(self, query_set, relevant_docs):
        """优化混合检索的权重"""
        best_weights = {"sparse": 0.5, "dense": 0.5}
        best_f1 = 0

        for sparse_weight in np.arange(0.1, 1.0, 0.1):
            dense_weight = 1 - sparse_weight
            # self.vectordb.update_weights(sparse_weight, dense_weight) #
Assuming SimpleVectorDB has an update_weights method

            f1_scores = []
            for query, relevant in zip(query_set, relevant_docs):
                # Pass the dense_encoder to the search method
                retrieved = self.vectordb.search(query, topk=3,
dense_encoder=self.vectordb.dense_encoder)
                eval_results = evaluate_retrieval_performance(
                    [doc for doc, _ in retrieved],
                    relevant
                )
                f1_scores.append(eval_results["f1"])

            avg_f1 = np.mean(f1_scores)
            if avg_f1 > best_f1:
                best_f1 = avg_f1
                best_weights = {"sparse": float(sparse_weight), "dense":
float(dense_weight)}

        return best_weights
```

```
    def optimize_model_inference(self):
        """优化模型推理性能"""
        # 简化模型优化，避免可能导致崩溃的量化操作
        logging.info("跳过量化，仅进行批处理优化")
        try:
            self.batch_size = self._find_optimal_batch_size()
            logging.info(f"最优批处理大小: {self.batch_size}")
        except Exception as e:
            logging.warning(f"批处理优化失败: {str(e)}")
            self.batch_size = 1

    def _find_optimal_batch_size(self, start_size=1, max_size=8):
        """找到最优的批处理大小，使用更安全的参数范围"""
        # 简化为固定批处理大小，避免复杂测试导致崩溃
        return 1  # 返回安全的批处理大小

def update_knowledge_base(feedback_data, vectordb):
    """
    基于用户反馈更新知识库

    参数:
        feedback_data (List[Dict]): 用户反馈数据
        vectordb: 向量数据库实例
    """
    for feedback in feedback_data:
        if feedback["rating"] < 3:  # 对于低评分的回答
            # 分析错误原因
            error_type = analyze_error(feedback["query"],
feedback["answer"])

            # 更新知识库
            if error_type == "missing_information":
                # 添加新的知识条目
                new_doc = generate_knowledge_entry(feedback["query"],
feedback["correct_answer"])
                # vectordb.add_document(new_doc) # Assuming SimpleVectorDB
has an add_document method
            elif error_type == "outdated_information":
                # 更新过时的知识条目
                update_existing_entry(feedback["query"],
feedback["correct_answer"], vectordb)

def analyze_error(query, answer):
```

```
    """分析错误类型"""
    # 实现错误分析逻辑
    pass

def generate_knowledge_entry(query, correct_answer):
    """生成新的知识条目"""
    # 实现知识条目生成逻辑
    pass

def update_existing_entry(query, correct_answer, vectordb):
    """更新已有的知识条目"""
    # 实现知识条目更新逻辑
    pass
'''''
# ========== 主流程 ==========
def main():
    print("欢迎使用 RAG 电商知识库问答系统，直接输入你的问题（输入 exit/退出/空行结束）：")
    # 检查 CUDA
    assert torch.cuda.is_available(), "需要 CUDA 支持"
    device = torch.device("cuda:0")

    # 初始化日志
    logging.info(f"加载知识库与检索器...")

    # 初始化性能监控和优化管理器
    performance_monitor = PerformanceMonitor()

    # 数据预处理
    docs = RAW_DOCS
    chunks = chunk_docs(docs)
    tfidf_vectorizer, tfidf_matrix = build_tfidf(chunks)

    #
================================================================================
====================================
    #
================================================================================
====================================
    # 在这里修改不同的 embedding 模型
    dense_encoder = build_dense_encoder(1)
    #
================================================================================
====================================
```

```
    #
==========================================================================
====================================


    dense_embeds = encode_dense(dense_encoder, chunks, is_query=False)
    vectordb = SimpleVectorDB(chunks, tfidf_matrix, tfidf_vectorizer,
dense_embeds, dense_encoder)

    # 加载生成模型
    logging.info(f"加载生成模型...")
    # model_path = '/home/mw/.cache/modelscope/hub/models/Qwen/Qwen3-4B'
    model_path = '/home/mw/input/qwen3_4B73447344'
    model, tokenizer = load_generation_model(model_path, device)

    # 初始化优化管理器
    optimization_manager = OptimizationManager(vectordb, model, tokenizer)

    # 优化模型推理性能（简化优化，避免内存问题）
    logging.info("正在优化模型推理性能...")
    try:
        optimization_manager.optimize_model_inference()
    except Exception as e:
        logging.error(f"优化模型推理性能失败: {str(e)}")

    # 初始化评估指标统计
    session_metrics = {
        "total_queries": 0,
        "successful_queries": 0,
        "avg_response_time": [],
        "quality_metrics": [],
        "retrieval_metrics": []  # 新增检索指标统计
    }

    # 定义优化触发条件
    OPTIMIZATION_INTERVAL = 100  # 每处理100个查询进行一次优化
    QUALITY_THRESHOLD = 0.7     # 质量指标阈值

    while True:
        query = input("\n请输入你的问题: ").strip()
        if query.lower() in ("exit", "退出", "quit", ""):
            # 输出会话统计信息
            if session_metrics["total_queries"] > 0:
                print("\n=== 会话统计 ===")
```

```
                print(f"总查询数: {session_metrics['total_queries']}")
                print(f"成功率:
{session_metrics['successful_queries']/session_metrics['total_queries']
:.2%}")
                print(f"平均响应时间:
{np.mean(session_metrics['avg_response_time']):.2f}秒")

                # 显示更详细的质量评估指标
                if session_metrics['quality_metrics']:
                    avg_factual = np.mean([m['factual_accuracy'] for m in
session_metrics['quality_metrics']])
                    avg_relevance = np.mean([m['context_relevance'] for m in
session_metrics['quality_metrics']])
                    avg_hallucination =
np.mean([m.get('hallucination_rate', 0) for m in
session_metrics['quality_metrics']])
                    avg_perplexity = np.mean([m.get('perplexity', 50.0) for
m in session_metrics['quality_metrics']])

                    print("\n=== 生成质量评估 ===")
                    print(f"平均事实准确性: {avg_factual:.2f} (越高越好)")
                    print(f"平均上下文相关性: {avg_relevance:.2f} (越高越
好)")
                    print(f"平均幻觉率: {avg_hallucination:.2f} (越低越好)")
                    print(f"平均困惑度: {avg_perplexity:.2f} (越低越好)")

                    # 添加 BLEU 分数显示
                    if NLTK_AVAILABLE:
                        avg_bleu = np.mean([m.get('bleu_score', 0) for m in
session_metrics['quality_metrics']])
                        print(f"平均 BLEU 分数: {avg_bleu:.4f} (越高越好)")

                    # 整体质量评估
                    # 归一化各指标，并计算加权平均
                    norm_factual = avg_factual  # 已经在 0-1 范围
                    norm_relevance = avg_relevance  # 已经在 0-1 范围
                    norm_hallucination = 1 - avg_hallucination  # 转换为正
向指标
                    norm_perplexity = 1 - (avg_perplexity / 50.0)  # 转换为
0-1 的正向指标

                    overall_quality = 0.3 * norm_factual + 0.3 *
norm_relevance + 0.2 * norm_hallucination + 0.2 * norm_perplexity
```

```
                    print(f"整体答案质量: {overall_quality:.2f} (0-1 分，越
高越好)")
                else:
                    print("未收集到质量评估指标")

                # 显示检索评估指标
                if session_metrics['retrieval_metrics']:
                    avg_precision = np.mean([m['precision'] for m in
session_metrics['retrieval_metrics']])
                    avg_recall = np.mean([m['recall'] for m in
session_metrics['retrieval_metrics']])
                    avg_f1 = np.mean([m['f1'] for m in
session_metrics['retrieval_metrics']])
                    avg_mrr = np.mean([m.get('mrr', 0) for m in
session_metrics['retrieval_metrics']])
                    print(f"\n=== 检索评估指标 ===")
                    print(f"平均精确率(Precision): {avg_precision:.2f}")
                    print(f"平均召回率(Recall): {avg_recall:.2f}")
                    print(f"平均 F1 分数: {avg_f1:.2f}")
                    print(f"平均排名(MRR): {avg_mrr:.2f}")

                # 输出性能统计
                perf_stats = performance_monitor.get_statistics()
                print("\n=== 性能统计 ===")
                print(f"P95 响应时间: {perf_stats.get('p95_response_time',
0):.2f}秒")
                print(f"平均 GPU 使用: {perf_stats.get('avg_gpu_usage',
0):.2f}MB")
                if perf_stats.get('error_statistics'):
                    print("错误统计:", perf_stats['error_statistics'])

                # 检索性能统计
                print("\n=== 检索性能 ===")
                print(f"平均检索时间: {np.mean([t for t in
session_metrics['avg_response_time']]):.4f}秒")

                # 显示优化后的检索权重
                try:
                    retrieval_stats = {"检索策略": "混合检索 (TF-IDF + 向量
嵌入)"}
                    print(f"检索策略: {retrieval_stats['检索策略']}")
                    print(f"当前检索权重配置 - 稀疏: 0.1, 密集: 0.9")

                    # 尝试分析检索质量
```

```
                if len(session_metrics['quality_metrics']) > 0:
                    retrieved_doc_count = 3  # 默认每次检索 3 个文档
                    total_queries = session_metrics['total_queries']
                    print(f"检索文档总数: {retrieved_doc_count *
total_queries}")
                    print(f"检索结果平均相关性: {avg_relevance:.2f}")
            except Exception as e:
                logging.warning(f"显示检索性能统计时出错: {str(e)}")

            # 生成性能统计
            try:
                print("\n=== 生成性能 ===")
                print(f"平均生成时间: {np.mean([t for t in
session_metrics['avg_response_time']]):.4f}秒")
                print(f"批处理大小: {getattr(optimization_manager,
'batch_size', 1)}")

                # 生成质量评估
                if len(session_metrics['quality_metrics']) > 0:
                    print(f"生成答案准确率: {avg_factual:.2f}")
                    print(f"生成模型: Qwen3-4B")
                    print(f"最大生成长度: 256")  # 使用常量值，与
generate_answer 函数默认值一致
            except Exception as e:
                logging.warning(f"显示生成性能统计时出错: {str(e)}")

        break

    # 记录查询开始时间
    start_time = time.time()

    try:
        session_metrics["total_queries"] += 1

        # 身份声明自动应答
        if check_identity_question(query):
            answer = IDENTITY_ANSWER + query + '\"'
            print(answer)
            continue

        # 检索相关文档
        logging.info(f"检索相关文档...")

        # 使用 SimpleVectorDB 的 search 方法进行混合检索
```

```
            retrieved = vectordb.search(query, topk=3, mode="hybrid",
dense_encoder=dense_encoder)

            # 构建上下文
            context = '\n'.join([x[0] for x in retrieved])

            print("\n【检索到的相关知识片段】")
            for i, (txt, score) in enumerate(retrieved):
                print(f"[{i+1}] {txt} (score={score:.4f})")

            # 生成答案
            logging.info(f"生成答案...")
            answer = generate_answer(model, tokenizer, context, query,
device)

            # 评估答案质量
            quality_metrics = evaluate_answer_quality(answer, [doc for doc,
_ in retrieved], query)
            session_metrics["quality_metrics"].append(quality_metrics)

            # 评估检索性能（简化示例，实际应用中需要真实标注数据）
            # 这里假设所有检索到的文档都是相关的，仅作为示例
            retrieval_metrics = evaluate_retrieval_performance(
                [doc for doc, _ in retrieved],  # 检索到的文档
                [doc for doc, _ in retrieved]   # 假设这些就是相关文档（实际
应用中需要真实标注）
            )
            session_metrics["retrieval_metrics"].append(retrieval_metric
s)

            # 记录成功查询
            session_metrics["successful_queries"] += 1

            print("\n【智能助手回答】\n" + answer)

            # 记录响应时间
            end_time = time.time()
            response_time = end_time - start_time
            session_metrics["avg_response_time"].append(response_time)
            performance_monitor.record_response_time(start_time,
end_time)

            # 记录资源使用
            performance_monitor.record_resource_usage()
```

```
            # 检查是否需要优化（增加错误处理）
            try:
                if (session_metrics["total_queries"] %
OPTIMIZATION_INTERVAL == 0 or
                    quality_metrics["context_relevance"] <
QUALITY_THRESHOLD):
                    logging.info("触发系统优化...")
                    # 优化检索权重
                    best_weights =
optimization_manager.optimize_retrieval_weights(
                        [query], [[doc for doc, _ in retrieved]]
                    )
                    # 确保返回的权重是标准 Python 类型，而不是 numpy 类型
                    best_weights = {k: float(v) if hasattr(v, 'item') else
v for k, v in best_weights.items()}
                    logging.info(f"更新检索权重: {best_weights}")

                    # 优化模型推理（简化优化过程）
                    try:
                        optimization_manager.optimize_model_inference()
                    except Exception as e:
                        logging.warning(f"模型推理优化失败: {str(e)}")
            except Exception as e:
                logging.error(f"系统优化过程发生错误: {str(e)}")

        except Exception as e:
            logging.error(f"处理查询时发生错误: {str(e)}")
            performance_monitor.record_error(type(e).__name__)
            print(f"\n抱歉，处理您的问题时遇到了错误: {str(e)}")
            continue
```

进阶作业：

```
# your code
import faiss
import numpy as np
import os
import time
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity
from sentence_transformers import SentenceTransformer

# ========== 数据收集与预处理 ==========
```

```
# 电商知识文档
RAW_DOCS = [
    # 商品信息
    "商品 A：高性能笔记本电脑，16GB 内存，512GB SSD，适合办公与游戏，售价 5999 元，支持分期付款。",
    "商品 B：无线蓝牙耳机，降噪功能，续航 30 小时，适合运动与通勤，售价 499 元，赠送收纳盒。",
    "商品 C：智能手表，支持心率监测、睡眠分析、运动追踪，防水 50 米，续航 7 天，售价 1299 元。",
    "商品 D：家用智能扫地机器人，激光导航，APP 控制，自动回充，适合各种地板清洁，售价 2499 元。",
    "商品 E：专业级数码相机，2400 万像素，4K 视频拍摄，防抖功能，含 18-55mm 标准镜头，售价 6299 元。",
    "商品 F：便携式蓝牙音箱，360°环绕立体声，防水防尘，续航 12 小时，支持 TWS 双音箱连接，售价 299 元。",
    "商品 G：多功能料理机，搅拌、切碎、榨汁多合一，2000W 大功率，8 档调速，静音设计，售价 899 元。",

    # 促销政策
    "促销政策：满 1000 减 100，部分商品参与，详情请咨询客服。",
    "限时特惠：每日 10 点、14 点、20 点开启秒杀，低至 5 折，每人限购 1 件。",
    "会员专享：银卡会员 95 折，金卡会员 9 折，钻石会员 85 折，不与其他优惠同享。",
    "新人福利：首次下单立减 50 元，无门槛使用，有效期 7 天。",
    "节日活动：618 年中大促，全场商品满减，部分商品买二送一。",
    "积分兑换：消费 1 元积 1 分，积分可兑换优惠券或实物礼品，详见积分商城。",
    "优惠券规则：优惠券不可叠加使用，不可与满减活动同享，有效期请见券面说明。",

    # 售后服务
    "售后服务：7 天无理由退换货，1 年质保，支持全国联保。",
    "退货政策：商品签收后 7 天内可申请无理由退货，商品需保持原包装及完好，退回运费由买家承担。",
    "换货流程：联系客服提交换货申请，审核通过后寄回商品，收到退回商品后 3 个工作日内发出新商品。",
    "保修条款：电子产品享受 1 年免费保修，人为损坏、擅自拆机、进水或改装不在保修范围内。",
    "售后网点：全国设有 1000+售后服务网点，可提供上门维修或到店维修服务。",
    "延长保修：可购买延长保修服务，最多可延长至 3 年，费用为商品价格的 5%-10%。",

    # 物流配送
    "物流说明：订单 24 小时内发货，支持多家快递，包邮服务。",
    "配送范围：全国大部分地区支持配送，港澳台及偏远地区可能产生额外运费。",
    "运费规则：单笔订单满 99 元免运费，不满 99 元收取 10 元运费，特大件商品另计。",
```

```
    "发货时间：工作日 16 点前下单当天发货，节假日及特殊情况可能顺延。",
    "自提服务：支持就近门店自提，下单时选择自提点，收到提货通知后凭码取货。",
    "极速达：部分城市支持 2 小时极速达服务，订单满足条件可在下单页面选择。",

    # 支付方式
    "支付方式：支持支付宝、微信支付、银联、信用卡等多种支付方式。",
    "分期付款：单笔订单满 500 元可申请 3-24 期分期，部分银行卡用户可享免息特权。",
    "货到付款：特定区域支持货到付款服务，需支付 5 元手续费。",
    "发票开具：可开具电子发票或纸质发票，请在下单时选择，纸质发票将随商品一起寄出。",

    # 会员体系
    "会员等级：普通会员、银卡会员（累计消费 5000 元）、金卡会员（累计消费 20000 元）、钻石会员（累计消费 50000 元）。",
    "会员权益：专属客服、生日礼遇、提前购、专享折扣、积分加速、免费试用等，等级越高权益越多。",
    "积分规则：消费 1 元获得 1 积分，参与活动可获得额外积分，积分有效期为一年。",

    # 常见问题
    "订单修改：订单支付成功后，如需修改收货信息请立即联系客服，发货后无法修改。",
    "账户安全：定期修改密码，不要在不信任的设备上登录账号，警惕钓鱼网站和诈骗信息。",
    "商品咨询：关于商品参数、适用场景等问题可咨询在线客服或拨打服务热线 400-888-8888。",
    "投诉建议：对服务不满意可通过 APP 意见反馈或发送邮件至 service@example.com 进行投诉。"
]

# 文档分块函数
def chunk_docs(docs, chunk_size=50, overlap=10):
    chunks = []
    for doc in docs:
        start = 0
        while start < len(doc):
            end = min(start + chunk_size, len(doc))
            chunk = doc[start:end]
            chunks.append(chunk)
            if end == len(doc):
                break
            start += chunk_size - overlap
    return chunks
```

```python
# ========== 向量数据库实现 ==========
# 原始向量数据库 (SimpleVectorDB)
class SimpleVectorDB:
    def __init__(self):
        self.vectors = None
        self.texts = []
        self.tfidf_vectorizer = TfidfVectorizer()
        self.tfidf_matrix = None

    def add(self, vectors, texts):
        """添加向量和文本"""
        if self.vectors is None:
            self.vectors = vectors
        else:
            self.vectors = np.vstack([self.vectors, vectors])

        self.texts.extend(texts)
        self.tfidf_matrix =
self.tfidf_vectorizer.fit_transform(self.texts)

    def search(self, query_vector, topk=3, mode="dense"):
        """支持稠密向量和稀疏向量检索"""
        if mode == "dense":
            similarities = cosine_similarity([query_vector],
self.vectors)[0]
        elif mode == "sparse":
            q_vec = self.tfidf_vectorizer.transform([query_text])
            similarities = cosine_similarity(q_vec, self.tfidf_matrix)[0]
        else:  # hybrid
            q_vec = self.tfidf_vectorizer.transform([query_text])
            sparse_sim = cosine_similarity(q_vec, self.tfidf_matrix)[0]
            dense_sim = cosine_similarity([query_vector], self.vectors)[0]
            similarities = 0.5 * sparse_sim + 0.5 * dense_sim

        top_indices = np.argsort(similarities)[::-1][:topk]
        return [(self.texts[i], similarities[i]) for i in top_indices]

    def save(self, path):
        """保存到磁盘"""
        np.savez(path, vectors=self.vectors, texts=self.texts)

    def get_resource_stats(self, save_path):
        """获取资源占用统计"""
        # 内存占用 (MB)
```

```
        mem_usage = self.vectors.nbytes / 1024 / 1024 if self.vectors is not
None else 0

        # 磁盘占用 (MB)
        disk_usage = os.path.getsize(save_path) / 1024 / 1024 if
os.path.exists(save_path) else 0

        return {
            "memory_mb": mem_usage,
            "disk_mb": disk_usage
        }

# FAISS 向量数据库
class FAISSVectorDB:
    def __init__(self, dimension, index_type="IVF_FLAT"):
        self.dimension = dimension
        self.index_type = index_type
        self.texts = []
        self.nlist = None

        supported_types = ["FLAT", "IVF_FLAT"]
        if index_type not in supported_types:
            print(f"警告：不支持的索引类型{index_type}，自动切换为 IVF_FLAT")
            self.index_type = "IVF_FLAT"

        # 初始化索引
        if index_type == "FLAT":
            self.index = faiss.IndexFlatL2(dimension)
        elif index_type == "IVF_FLAT":
            self.quantizer = faiss.IndexFlatL2(dimension)
            self.index = None  # 延迟初始化

    def train(self, vectors):
        """训练索引（动态设置 nlist）"""
        if self.index_type == "IVF_FLAT":
            # 动态计算 nlist：确保 nlist ≤ 样本数
            self.nlist = min(max(10, vectors.shape[0] // 5),
vectors.shape[0])
            print(f"根据样本数{vectors.shape[0]}自动设置聚类中心数
nlist={self.nlist}")

            # 初始化 IVF 索引
            self.index = faiss.IndexIVFFlat(
                self.quantizer,
```

```
                self.dimension,
                self.nlist,
                faiss.METRIC_L2
            )

            # 训练索引
            if not self.index.is_trained:
                self.index.train(vectors)

    def add(self, vectors, texts):
        """添加向量和文本"""
        self.index.add(vectors)
        self.texts.extend(texts)

    def search(self, query_vector, topk=3):
        """检索最相似的向量"""
        distances, indices = self.index.search(query_vector.reshape(1, -1),
topk)
        # 将距离转换为相似度
        max_dist = np.max(distances) if np.max(distances) > 0 else 1
        similarities = 1 - (distances / max_dist)
        return [(self.texts[i], similarities[0][j]) for j, i in
enumerate(indices[0])]

    def save(self, path):
        """保存索引到磁盘"""
        faiss.write_index(self.index, path)
        np.savez(f"{path}_texts", texts=self.texts)

    def get_resource_stats(self, index_path):
        """获取资源占用统计"""
        vec_count = self.index.ntotal
        vec_memory = vec_count * self.dimension * 4 / 1024 / 1024  # float32
占 4 字节

        index_memory = 0
        if self.index_type == "IVF_FLAT":
            index_memory = self.nlist * self.dimension * 4 / 1024 / 1024

        index_disk = os.path.getsize(index_path) / 1024 / 1024 if
os.path.exists(index_path) else 0
        texts_disk = os.path.getsize(f"{index_path}_texts.npz") / 1024 /
1024 if os.path.exists(f"{index_path}_texts.npz") else 0
```

```
        return {
            "memory_mb": vec_memory + index_memory,
            "disk_mb": index_disk + texts_disk
        }

# ========== 评估函数 ==========
def evaluate_keyword_matching(retrieved_chunks, query):
    """评估检索结果中是否包含查询的核心关键词"""
    # 电商场景核心关键词库
    keyword_map = {
        "退货": ["退货", "7 天无理由", "退换货"],
        "商品 A 的价格与性能": ["商品 A", "5999 元", "16GB 内存", "游戏", "办公
"],
        "钱不够怎么办": ["分期付款", "免息", "优惠券", "满减"],
        "会员有什么优惠": ["银卡", "金卡", "钻石", "折扣", "95 折", "9 折", "85
折"],
        "多久能发货": ["24 小时", "发货", "工作日", "16 点前"]
    }
    query_keywords = keyword_map.get(query, [])
    if not query_keywords:
        return 0.0

    # 统计检索结果中匹配的关键词比例
    matched = 0
    for kw in query_keywords:
        for chunk, _ in retrieved_chunks:
            if kw in chunk:
                matched += 1
                break
    return matched / len(query_keywords)

def test_vector_db_performance(db, query_vectors, queries, topk=3):
    """测试向量数据库的检索性能和准确性"""
    start_time = time.time()
    keyword_scores = []

    # 多次检索
    for q_vec, query_text in zip(query_vectors, queries):
        retrieved = db.search(q_vec, topk=topk)
        keyword_scores.append(evaluate_keyword_matching(retrieved,
query_text))

    avg_time = (time.time() - start_time) / len(query_vectors) * 1000
    avg_keyword_score = sum(keyword_scores) / len(keyword_scores)
```

```
    return {
        "avg_time_ms": avg_time,
        "avg_keyword_score": avg_keyword_score
    }

# ========== 流程 ==========
def main():
    # 初始化配置
    SAVE_PATHS = {
        "simple": "simple_vector_db.npz",
        "faiss_flat": "faiss_flat_index.index",
        "faiss_ivf": "faiss_ivf_index.index",
        "faiss_ivf_sq8": "faiss_ivf_sq8_index.index"
    }

    print("加载 Qwen3-Embedding-0.6B 模型...")
    embedding_model =
SentenceTransformer("/home/mw/.cache/modelscope/hub/models/Qwen/Qwen3-E
mbedding-0.6B")
    VECTOR_DIM = embedding_model.get_sentence_embedding_dimension()  # 获
取实际向量维度
    print(f"向量维度: {VECTOR_DIM}")

    print("\n 处理电商知识库数据...")
    chunks = chunk_docs(RAW_DOCS, chunk_size=50, overlap=10)
    print(f"原始文档 {len(RAW_DOCS)} 条，分块后得到 {len(chunks)} 个文本块")

    print("生成文本向量...")
    vectors = embedding_model.encode(chunks,
convert_to_numpy=True).astype('float32')

    # 测试查询
    test_queries = [
        "退货",
        "商品 A 的价格与性能",
        "钱不够怎么办",
        "会员有什么优惠",
        "多久能发货"
    ]
    query_vectors = embedding_model.encode(test_queries,
convert_to_numpy=True).astype('float32')

    results = {}
```

```
    # 测试原始向量数据库 (SimpleVectorDB)
    print("\n===== 测试原始向量数据库 (SimpleVectorDB) =====")
    simple_db = SimpleVectorDB()
    simple_db.add(vectors, chunks)
    simple_db.save(SAVE_PATHS["simple"])

    simple_perf = test_vector_db_performance(simple_db, query_vectors,
test_queries)
    simple_stats = simple_db.get_resource_stats(SAVE_PATHS["simple"])

    results["SimpleVectorDB"] = {
        "avg_time_ms": simple_perf["avg_time_ms"],
        "memory_mb": simple_stats["memory_mb"],
        "disk_mb": simple_stats["disk_mb"],
        "keyword_score": simple_perf["avg_keyword_score"]
    }

    print(f"平均检索时间: {simple_perf['avg_time_ms']:.2f} ms")
    print(f"内存占用: {simple_stats['memory_mb']:.2f} MB")
    print(f"磁盘占用: {simple_stats['disk_mb']:.2f} MB")
    print(f"关键词匹配率: {simple_perf['avg_keyword_score']:.2f}")

    # 测试 FAISS (Flat 索引)
    print("\n===== 测试 FAISS (Flat 索引) =====")
    faiss_flat_db = FAISSVectorDB(dimension=VECTOR_DIM,
index_type="FLAT")
    faiss_flat_db.add(vectors, chunks)
    faiss_flat_db.save(SAVE_PATHS["faiss_flat"])

    faiss_flat_perf = test_vector_db_performance(faiss_flat_db,
query_vectors, test_queries)
    faiss_flat_stats =
faiss_flat_db.get_resource_stats(SAVE_PATHS["faiss_flat"])

    results["FAISS (Flat)"] = {
        "avg_time_ms": faiss_flat_perf["avg_time_ms"],
        "memory_mb": faiss_flat_stats["memory_mb"],
        "disk_mb": faiss_flat_stats["disk_mb"],
        "keyword_score": faiss_flat_perf["avg_keyword_score"]
    }

    print(f"平均检索时间: {faiss_flat_perf['avg_time_ms']:.2f} ms")
    print(f"内存占用: {faiss_flat_stats['memory_mb']:.2f} MB")
```

```
    print(f"磁盘占用: {faiss_flat_stats['disk_mb']:.2f} MB")
    print(f"关键词匹配率: {faiss_flat_perf['avg_keyword_score']:.2f}")

    # 测试 FAISS (IVF_FLAT 索引)
    print("\n===== 测试 FAISS (IVF_FLAT 索引) =====")
    faiss_ivf_db = FAISSVectorDB(dimension=VECTOR_DIM,
index_type="IVF_FLAT")
    faiss_ivf_db.train(vectors)  # IVF 需要训练
    faiss_ivf_db.add(vectors, chunks)
    faiss_ivf_db.save(SAVE_PATHS["faiss_ivf"])

    faiss_ivf_perf = test_vector_db_performance(faiss_ivf_db,
query_vectors, test_queries)
    faiss_ivf_stats =
faiss_ivf_db.get_resource_stats(SAVE_PATHS["faiss_ivf"])

    results["FAISS (IVF_FLAT)"] = {
        "avg_time_ms": faiss_ivf_perf["avg_time_ms"],
        "memory_mb": faiss_ivf_stats["memory_mb"],
        "disk_mb": faiss_ivf_stats["disk_mb"],
        "keyword_score": faiss_ivf_perf["avg_keyword_score"]
    }

    print(f"平均检索时间: {faiss_ivf_perf['avg_time_ms']:.2f} ms")
    print(f"内存占用: {faiss_ivf_stats['memory_mb']:.2f} MB")
    print(f"磁盘占用: {faiss_ivf_stats['disk_mb']:.2f} MB")
    print(f"关键词匹配率: {faiss_ivf_perf['avg_keyword_score']:.2f}")


    # 对比表格
    print("\n===== 向量数据库综合对比结果 =====")
    print(f"| 向量数据库类型       | 平均检索时间(ms) | 内存占用(MB) | 磁盘占
用(MB) | 关键词匹配率 |")
    print(f"|----------------------|------------------|--------------|--
------------|--------------|")
    for name, stats in results.items():
        print(f"| {name:<20} | {stats['avg_time_ms']:<16.2f} |
{stats['memory_mb']:<12.2f} | {stats['disk_mb']:<12.2f} |
{stats['keyword_score']:<12.2f} |")

if __name__ == "__main__":
    main()
```