大数据系统及综合实验期末大作业：Search Engine

申长硕 @PB22020518　　高茂航 @PB22061161
AI&DS
{stephen_shen, gmh1627} @mail.ustc.edu.cn

2024 年 12 月 25 日

摘要

本实验中我们构建了一个面向 USTC 各网站的文档检索系统，通过自动化爬取获取关键文档，并基于 TF-IDF 方法实现相关性排序，同时在前端提供可视化的查询与结果展示界面，尽可能做到用户友好。实验中我们还尝试结合本地部署的大语言模型 ChatGLM，借助 RAG（Retrieval Augmented Generation）技术增强搜索结果，为用户提供更好的交互式问答体验。代码开源在：https://github.com/ChangshuoShen/Search-Engine

1　背景与目标

1.1　研究背景

随着信息化时代的到来，学校网站成为了重要的资源和信息发布平台，其中包含大量学术和行政文件。如果没有一个良好的搜索引擎进行检索，教师与学生在需要查找资料时往往耗时较长。为了提高文档资源的访问效率，我们亟需一个面向 USTC 各网站的统一文档检索系统。

1.2　主要目标

本实验旨在完成以下任务：

- **构建较大规模的校内文档库：**通过自动爬取并收集来自 USTC 各学院与管理部门官网的文件和文本数据，包括但不限于学术论文、规章制度、会议通知等。
- **实现高效检索：**使用 TF-IDF 等文本检索技术，在“海量”文档中迅速找到与用户查询相关的内容，并根据相关度进行排序。
- **前端展示与交互：**借助 Flask 搭建一个可视化的检索系统，并通过 RAG 技术与本地部署的 ChatGLM 接口，增强搜索体验，为用户提供检索结果与智能问答。
- **应用大数据相关技术：**用到 HBase 分布式数据库、爬虫脚本、文本分析与关键词提取等，运用本课程中所学到的大数据系统和数据处理技术。

2　技术路线

2.1　整体技术框架

整体流程：

1. **爬虫与数据获取：**从 USTC 各个网站自动化爬取文件和文本，重点关注“下载中心”或有大量文件链接的板块。

2. **数据预处理与分析：**对爬取下来的 HTML、PDF、DOC 等文件进行格式转换或解析，使用 TF-IDF 提取关键词，同时记录高频词以辅助检索。
3. **数据存储：**将文件的元数据（关键词、高频词、文件路径等）存入 HBase，部分文件内容也可做分词或索引处理后存入数据库。
4. **检索服务：**根据用户输入，通过匹配文件标题、关键词和高频词等信息，计算与查询的相似度，并将最相关的文件按分值排序返回。
5. **前端与可视化：**使用 Flask 搭建前端，将搜索结果以列表形式呈现，并支持与 ChatGLM 的问答接口。

2.2 关键技术模块

- **爬虫技术：**采用 Python 脚本 Requests+lxml 或 Requests+BeautifulSoup 等方法，根据预先收集的链接列表递归爬取子链接，下载文件。
- **分词与关键词抽取：**基于 Python 的 jieba 分词，结合 TF-IDF 算法提取文档的关键主题词。
- **HBase：**通过 `happybase`，将关键词索引信息以及文件路径写入到 HBase 中实现快速检索。
- **检索算法：**主要使用 TF-IDF 方案进行相关度计算，(同时可考虑向量空间模型、BM25 或其他加权方式进一步提升检索效果，这里本文没有用到)。
- **RAG + ChatGLM：**在用户搜索结果的基础上，利用检索到的文本作为提示上下文，对接本地 ChatGLM API，实现回复的内容更加准确、更具上下文相关性。(本实验在此处仅提供框架，未调试性能)
- **Flask 前端：**提供简单友好的用户界面，包括查询输入框、搜索结果展示列表，以及与 ChatGLM 的对话体验。

3 方法与实现

3.1 分工说明

- **高茂航：**部分爬虫脚本、数据清理与部分测试、课堂展示。
- **申长硕：**部分爬虫脚本、全文档关键词提取、HBase 环境配置、数据导入与检索、Flask 前端界面开发、ChatGLM 接口整合、实验报告的撰写。

3.2 项目结构

实验项目的代码仓库：https://github.com/ChangshuoShen/Search-Engine整体结构如下所示：

Listing 1: 项目代码结构

```
.
    app
        __init__.py
        routes.py
        static
        templates
    cache
        advanced
        bio
```

```
        bme
    ...
        zhc
    chat
        chatglm_api.py
    crawl
        GMH
        Shencs
        temp.py
        url_gmh.yml
        url_scs.yml
    crawl.sh
    database
        const.py
        hbase.py
        hdfs.py
        search.py
        urlcontent.py
        utils.py
    keywords
        file_keywords.json
        folder_keywords.json
        keywords_each_file.py
        keywords.py
        stopwords.txt
    scripts
        test_search.py
        upload2database.py
        upload2hbase.py
    run.py
    大数据系统-实验文档.pdf
```

以下分别对各模块进行说明：

- `app/`：基于 Flask 实现的前端与路由逻辑，包括静态文件和模板页面。
- `cache/`：爬虫后缓存的文件夹，记录不同学院网站爬取结果，按学院或部门命名。
- `chat/`：封装了与本地部署 ChatGLM 模型的 API 接口，可对外提供基于 RAG 文档的问答功能。
- `crawl/`：包含所有爬虫脚本及配置文件（如 `url_gmh.yml`, `url_scs.yml`），用于批量抓取各类网站内容。
- `crawl.sh`：可执行脚本，用于批量调用爬虫脚本。
- `database/`：HBase、HDFS 等相关的数据库接口封装，包括连接、读写、查询功能等。本次实验主要使用 HBase，HDFS 因权限原因暂未启用。
- `keywords/`：关键词提取流程，包括 TF-IDF 计算、停用词（`stopwords.txt`）、结果输出等脚本。
- `scripts/`：一些辅助脚本，如向数据库批量上传文档、测试搜索性能等。
- `run.py`：启动整个项目的入口，通过 Flask 托管服务。

3.3 数据采集与爬虫

核心思路：

1. 根据预先整理的 USTC 各学院与管理部门链接列表，编写爬虫配置（.yml）文件。
2. 以 DFS 或 BFS 方式遍历网站链接，并过滤非本站链接。
3. 对页面中的文件（尤其是 PDF、DOC、DOCX、ZIP 等）链接进行下载，必要时进行简单分类存储。
4. 若网站包含“下载中心”或类似专栏，则重点扫描其中的链接。

示例：如爬虫脚本 finance.py 中核心逻辑：

Listing 2: 爬虫脚本示例

```python
import requests
from bs4 import BeautifulSoup
import os
import re

# 目标 URL 列表
urls = [
    'https://finance.ustc.edu.cn/xzzx/list.psp',
    'https://finance.ustc.edu.cn/xzzx/list2.psp',
    'https://finance.ustc.edu.cn/xzzx/list3.psp',
    'https://finance.ustc.edu.cn/xzzx/list4.psp',
    'https://finance.ustc.edu.cn/xzzx/list5.psp'
]

base_url = 'https://finance.ustc.edu.cn' # 用于处理相对路径

# 本地保存文件路径
save_path = 'cache/finance'
os.makedirs(save_path, exist_ok=True)

def fetch_page(url):
    """获取网页内容并解析为 BeautifulSoup 对象"""
    ...

def clean_filename(filename):
    """移除非法字符"""
    return re.sub(r'[\\/*?:"<>|]', "", filename)

def save_file(file_url, title, save_path):
    """下载并保存文件"""
    ...

def parse_detail_page(detail_url):
    """解析详情页，提取文件下载链接和标题"""
    ...

def parse_list_page(url):
    """解析列表页，提取文件下载链接或详情页链接"""
    ...
```

```
40
41 def crawl_site(url):
42     """爬取列表页并处理文件下载"""
43     ...
44
45 if __name__ == "__main__":
46     for url in urls:
47         crawl_site(url)
```

3.4 文本分析与关键词提取

对爬下来的文件，需要先进行格式转换或解析（如 PDF 转为文本、DOCX 提取文本等）。然后进行分词与 TF-IDF 计算。具体实现流程示意：

1. **文本解析：**判断文件类型，调用相应的 PDF 或 DOC 解析库（如 `pdfplumber`、`python-docx` 等）。
2. **分词处理：**将文本通过 jieba 进行分词，去除停用词（见 `keywords/stopwords.txt`）。
3. **TF-IDF 计算：**根据所有文档语料生成词频统计和逆文本频率，最终得到每个文档对应的关键词及其权重。
4. **记录高频词：**额外统计每个文档出现频率最高的若干词（如 top-10）以供检索时辅助排序。

示例代码

Listing 3: keywords_each_file.py

```
1  import os
2  import jieba
3  import fitz  # PyMuPDF，用于处理 PDF
4  from docx import Document  # 用于处理 DOCX
5  from sklearn.feature_extraction.text import TfidfVectorizer
6  from collections import Counter
7  import json
8
9  # 停用词表路径
10 STOPWORDS_PATH = "keywords/stopwords.txt"
11
12 def load_stopwords(path):
13     """加载停用词表"""
14     ...
15
16 def extract_text_from_pdf(file_path):
17     """从 PDF 文件中提取文本"""
18     ...
19
20 def extract_text_from_docx(file_path):
21     """从 DOCX 文件中提取文本"""
22     ...
23
24 def extract_text_from_txt(file_path):
25     """从 TXT 文件中提取文本"""
26     ...
27
```

```
def extract_text_from_file(file_path):
    """根据文件类型提取文本"""
    ...

def extract_keywords_and_high_freq_words(file_contents, stopwords, top_k=10, high_freq_n=5):
    """使用 TF-IDF 提取关键词，并从单个文档中提取高频词"""
    ...

def process_all_files(base_folder, output_file, top_k=10, high_freq_n=5):
    """扫描所有文件，提取每个文件的关键词和高频词"""
    ...


if __name__ == "__main__":
    base_folder = "cache/"  # 主文件夹路径
    output_file = "keywords/file_keywords.json"  # 输出文件路径
    top_k = 10  # TF-IDF 提取的关键词数量
    high_freq_n = 5  # 高频词数量

    process_all_files(base_folder, output_file, top_k, high_freq_n)
```

3.5 HBase 数据存储

将关键词和文档元数据信息写入 HBase。这里可以按照 `row_key` = 文件唯一标识（例如其在服务器的存储路径或 MD5），列簇可分为：

- **name:** 文件名。
- **keywords:** 存储提取到的关键词等。
- **freq:** 存储文件中高频词信息。

示例代码

Listing 4: database/hbase.py

```
import os
import happybase
import json
from typing import List, Dict
from .utils import get_logger
from .const import HBASE_HOST, HBASE_TABLE_NAME, HBASE_COLUMN_FAMILY, HDFS_TARGET_DIR,
    ↪ LOCAL_CACHE_DIR
from .urlcontent import URLContent


class USTCHBase:
    """
    封装 HBase 操作类。
    """
    def __init__(self, host=HBASE_HOST, column_family=HBASE_COLUMN_FAMILY):
        self.logger = get_logger('ustc-hbase')
        self.logger.info("HBase Connecting...")
```

```
        self.connection = happybase.Connection(host)
        self.connection.open()
        self.name = HBASE_TABLE_NAME
        if self.name.encode() not in self.connection.tables():
            assert column_family is not None, "Need to indicate column families to create a
                ↪ table"
            self.create_table(self.name, column_family)
        self.logger.info("HBase Connected Successfully!")
        self.table = self.get_table(self.name)

    def __enter__(self):
        return self

    def __exit__(self, type, message, traceback):
        ...

    def create_table(self, table_name: str, column_families: List[str]):
        ...

    def get_table(self, table_name: str):
        ...

    def put(self, row: bytes, data: Dict[bytes, bytes]):
        ...

    def query_meta(self, file_name: str):
        """根据文件名查询文件的元数据"""
        ...

    def scan(self):
        """扫描表中的所有数据"""
        ...

    def close_connection(self):
        self.connection.close()

def store_meta_in_hbase(json_file: str, hdfs_dir=HDFS_TARGET_DIR):
    """从 JSON 文件加载文件元数据，并存储到 HBase"""
    ...
```

将这些信息写入后，即可在检索时快速获取与查询相关的关键词和对应文档进行匹配。

3.6 检索引擎实现

整体思路：

1. 用户输入查询 query。
2. 对 query 进行分词，移除停用词（`stopwords.txt`）后，提取有效的关键词集合。
3. 使用分词后的查询词集合，与 HBase 中每个文档的 **标题（Title）**、**关键词（Keywords）** 和 **高频词（High Frequency Words）** 进行匹配。

4. 通过简单的指标（如 **交集大小**和 **IoU**）计算每个字段的相关性得分，并根据字段权重（如标题权重较高）计算总分：

$$\text{Score} = w_1 \cdot \text{TitleMatch} + w_2 \cdot \text{KeywordsMatch} + w_3 \cdot \text{HighFreqMatch} \tag{1}$$

5. 按 Score 降序返回排名前 k 的文档。

Listing 5: database/search.py

```python
import jieba
from typing import List, Set
from .hbase import USTCHBase
from .const import SEARCH_TOP_K


class SearchEngine:
    """基于词集合匹配的简单搜索引擎，支持分词、关键词匹配和加权排序"""
    def __init__(self):
        self.hbase = USTCHBase()
        self.stopwords = self.load_stopwords("keywords/stopwords.txt")

    def load_stopwords(self, filepath: str) -> Set[str]:
        """加载停用词表"""
        ...

    def tokenize(self, text: str) -> List[str]:
        """对文本进行分词，并移除停用词"""
        ...

    def calculate_iou(self, query_set: Set[str], doc_set: Set[str]) -> float:
        """计算查询词集合与文档集合的 IoU（交并比）"""
        ...

    def calculate_match_score(self, query_set: Set[str], doc_set: Set[str]) -> int:
        """计算查询词集合与文档集合的交集大小"""
        ...

    def search(self, query: str) -> List[dict]:
        """
        根据查询词搜索文档标题、关键词和高频词，并加权排序
        Args:
            query (str): 用户的搜索关键词
        Returns:
            List[dict]: 按相关性排序的搜索结果
        """
        results = []
        query_tokens = set(self.tokenize(query))  # 查询词集合

        for _, data in self.hbase.scan():
            title = data.get("cf0:title", "")
            keywords = set(data.get("cf0:keywords", "").split(","))
            high_freq_words = set(data.get("cf0:high_freq_words", "").split(","))

```

```
            # 计算 IoU （可选，用于权重调整或额外排序依据）
            keywords_iou = self.calculate_iou(query_tokens, keywords)
            high_freq_iou = self.calculate_iou(query_tokens, high_freq_words)

            # 计算匹配得分
            title_score = self.calculate_match_score(query_tokens, set(self.tokenize(title)))
            keywords_score = self.calculate_match_score(
                query_tokens, keywords) + 0.1 * keywords_iou
            high_freq_score = self.calculate_match_score(
                query_tokens, high_freq_words) + 0.15 * high_freq_iou

            # 加权得分
            total_score = (
                title_score * 5 +  # 文档名权重较高
                keywords_score * 2 +  # 关键词权重
                high_freq_score * 1  # 高频词权重
            )
            if total_score > 0:
                results.append({
                    "title": title,
                    "keywords": list(keywords),
                    "high_freq_words": list(high_freq_words),
                    "hdfs_path": data.get("cf0:hdfs_path", ""),
                    "total_score": total_score,
                    "keywords_iou": keywords_iou,
                    "high_freq_iou": high_freq_iou
                })

        # 按总分排序
        results.sort(key=lambda x: x["total_score"], reverse=True)
        return results[:SEARCH_TOP_K]
```

3.7 前端与 RAG 增强

3.7.1 Flask 前端

使用 Flask 搭建一个简易的前端，包含：

- **搜索页面：**提供搜索框用于输入检索词；
- **结果页面：**展示检索排名靠前的文档列表，同时提供“查看详情”入口。

前端文件主要放置于 `app/templates/` 和 `app/static/` 目录下。在 `app/routes.py` 中编写路由，示例：

Listing 6: app/routes.py

```
main = Blueprint('main', __name__)
search_engine = SearchEngine()

@main.route("/", methods=["GET", "POST"])
def index():
    ...

```

```
@main.route("/download/<path:filepath>")
def download(filepath):
    """文件下载功能"""
    ...

@main.route("/preview/<path:filepath>")
def preview(filepath):
    """文件预览功能"""
    ...
```

3.7.2 RAG + ChatGLM

结合 Retrieval-Augmented Generation （RAG）方法，在与 ChatGLM 交互时，先进行检索以获取与用户询问相关的文档片段，再将其作为上下文或提示输入给模型，以输出更精准的答复。

- **检索模块：**基于搜索引擎返回最匹配的文档或文本片段。
- **融合输入：**将检索结果拼接成上下文段落，与用户问题一起传递给 ChatGLM 模型。
- **输出生成：**由 ChatGLM 模型生成最终回答，体现出对校内文档的引用与依据。

Listing 7: app/routes.py

```
# 设置 tokenizer
tokenizer = AutoTokenizer.from_pretrained("THUDM/glm-4-9b-chat", trust_remote_code=True)

# FastAPI 的地址
# FASTAPI_URL = "http://127.0.0.1:8000/generate"  # 假设 FastAPI 运行在本地

@main.route("/rag", methods=["POST"])
def rag():
    """
    通过接收用户的查询，并结合搜索结果构建新的 prompt，
    然后调用 FastAPI 生成回答（RAG）。
    """
    ...
        # 将检索结果和原始查询组成新的 prompt
        combined_prompt = f"用户查询：{query}\n相关文档：\n"
        for result in search_results:
            combined_prompt += f"- {result['content']}\n"  # 假设 result['content'] 是文档内容
        ...
```

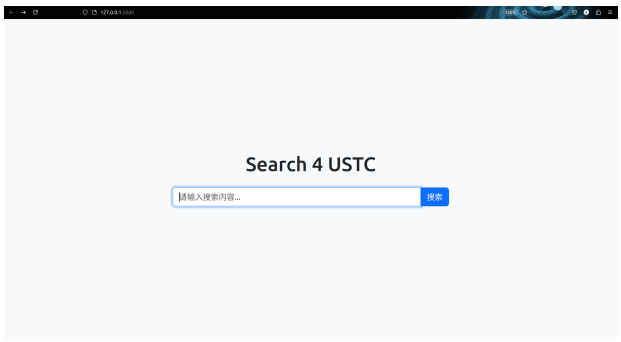
Listing 8: chat/chatglm_api.py

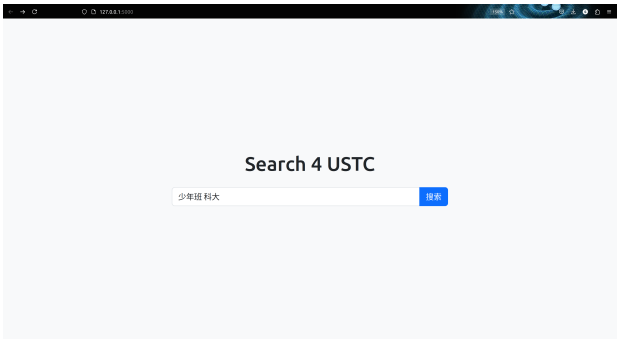
```
import torch
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel
from transformers import AutoModelForCausalLM, AutoTokenizer
import uvicorn

# 初始化 FastAPI 实例
app = FastAPI()

```

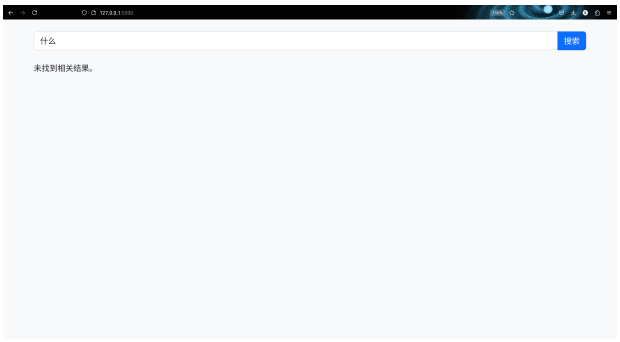
```
# 尝试加载模型和 tokenizer
try:
    ...

# 定义请求体格式
class QueryRequest(BaseModel):
    query: str

# 定义 API 路由，用于接收查询并返回生成的文本
@app.post("/generate")
async def generate_text(request: QueryRequest):
    """接收用户请求，使用 GLM 模型生成回答"""
    ...

# 在主程序中启动 API 服务
if __name__ == "__main__":
    # 使用 Uvicorn 启动 FastAPI 服务
    uvicorn.run(
        "chatglm_api:app",
        host="0.0.0.0",  # 监听所有网络接口
        port=8000, # 默认端口
        reload=True # 开启自动重载（注意这个是开发模式下使用）
    )
```

4 结果与分析

4.1 结果展示，实际效果可点击链接：search engine demo video 进行查看







4.2 实验环境与配置

- **操作系统：** Ubuntu 22.04；
- **Python 版本：** 3.9.21；
- **数据库：** HBase 2.1.1 (本地或伪分布模式)；
- **依赖库：** `Requests`，`BeautifulSoup`，`PyPDF2`，`python-docx`，`jieba`，`happybase`，`Flask` 等，具体可在 **requirements.txt** 中列出。

4.3 检索效果分析

- **爬取结果：** 成功爬取并下载了若干学院官网和管理部门官网的文件，包括 PDF、DOC、TXT、ZIP 等，约三千条文档。
- **查询测试：** 经过对常用关键词的测试（如“数学”、“财务报销”、“研究生培养方案”、“学术会议日程”等），系统能返回较为准确的文件链接及标题。

- **RAG 效果：**结合 ChatGLM 在检索结果中提炼信息，在 prompt 设计时包含检索到的文档内容，模型回答中也能体现出相关信息。
- **性能：**对常规查询能在数秒内得到结果，主要耗时在爬虫和文本解析阶段，但在实际检索中的响应较快。ChatGLM 模型在本人电脑上运行较慢，所以这里主要还是实现一个思路，写一个框架“玩玩”。

5 总结与反馈

5.1 踩坑

- **HBase 连接问题：**在本地伪分布模式下，出现连接超时或者 *Thrift* 服务未启动等问题，需要仔细检查防火墙设置、Thrift 端口是否开启。
- **文件解析失败：**部分 PDF 存在加密或格式不规范，导致解析库无法正确提取文本，需进行异常处理或者手动跳过。
- **编码与字符集：**由于文档中包含中文，爬虫和解析阶段容易出现编码混乱，需要统一使用 UTF-8 并进行必要的转码。
- **网络爬虫重定向问题：**某些学院官网频繁跳转，需要在爬虫脚本中对重定向进行处理，以免漏爬或出现无限循环。

5.2 错误总结

- **权限/防火墙：**在开发环境中使用 HDFS、HBase 时，需正确配置权限；否则会报错无法写入或读取。
- **依赖库版本冲突：**爬虫、文本解析、数据库等依赖库版本需尽量固定，否则可能因 API 变动引发兼容性问题。
- **表结构设计：**在 HBase 中未提前考虑列簇及 rowkey 的合理性，后期查询时可能遇到效率问题。

5.3 实验收获

- **综合运用：**通过本次实验，大家对 Python 爬虫、HBase、TF-IDF、Flask 前后端配合有了更深刻的理解。
- **可扩展性：**若未来需要扩大爬取范围或引入更多算法（如深度学习向量检索），我们目前的项目结构依然可复用。
- **协作与分工：**在小组合作中充分感受到协作的重要性，通过 Git 进行版本管理，提高了开发效率。

5.4 附录

代码仓库开源在：https://github.com/ChangshuoShen/Search-Engine