实验一报告

PB21000039 陈骆鑫，PB21000037 吴楚

第一阶段

爬虫

由于只找到了用于图书的API，这里图书使用API爬取，电影使用网页爬取。

图书爬取

根据豆瓣API文档留档，获取图书信息应当向 `https://api.douban.com/v2/book/:id` 发送GET请求。其返回的是包含作者、简介等大量信息的json文件，足够本次实验使用。

目前获取图书信息需要API key，这里使用了一个查找到的key，在请求后添加 `?apikey=...` 即可。

使用python语言中的requests库爬取。根据URL发送请求、接收信息的函数如下：

```
s = requests.session()

headers = {
    "User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/117.0.0.0 Safari/537.36"
}


def get_data(url):
    try:
        response = s.get(url, headers=headers, timeout=5)
        if response.status_code == 200:
            print("Success!")
            return json.loads(response.text)
        else:
            print("Failed!")
            print(response.text)
            if "book_not_found" in response.text:
                return "book_not_found"
            return None
    except requests.RequestException as e:
        print("Failed!")
        return None
```

在爬取主逻辑中，我们打开提供的id列表，对每行的id拼接URL，运行以上函数。注意存在极小部分id目前对应页面不存在，对这些id我们选择直接从id列表中删除。

对API爬取，豆瓣没有明显的反爬策略，但有一段时间内请求的数量限制，可以简单地在每次请求后暂停一段时间。

```
# crawl.py
with open("lab1/Book_id.csv", "r") as book_id:
    for line in book_id.readlines():
        id = line.rstrip()
        url = (
            "https://api.douban.com/v2/book/"
            + id
            + "?apikey=0ac44ae016490db2204ce0a042db2916"
        )
        print(url)
        path = "lab1/book_data/" + id + ".json"
        if os.path.exists(path):
            continue
        print("Try getting...")
        data = None
        while data is None:
            data = get_data(url)
            time.sleep(1)
        if data == "book_not_found":
            print("Book not found!")
            continue
        data = json.dumps(data, indent=4, ensure_ascii=False)
        save_data = open(path, "w", encoding="utf-8")
        save_data.write(data)
        time.sleep(1)
```

爬取得到的json已经是相当格式化的数据，无需进一步解析处理即可直接给下一部分使用。

电影爬取

没有找到能够使用的电影API key，因此电影的爬取使用网页爬取。网页爬取的基本任务是爬取页面 `https://movie.douban.com/subject/:id/` 的内容。

因此可以直接写出一个简单的脚本，执行对列表内所有id爬取网页的任务：

```
# movie_fetch.py
with open('lab1/Movie_id.csv') as file:
    flag = True
    str = file.readlines()
    for i in range(len(str)):
        id = str[i][:-1]
        path = 'lab1/movies/' + id
        if os.path.exists(path):
            continue
        flag = False
        url = 'https://movie.douban.com/subject/' + id
        print(url)
        headers = {'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64;
x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/79.0.3945.130
Safari/537.36 OPR/66.0.3515.115'}
```

```
            responce = requests.get(url, headers = headers)
            content = responce.content.decode('utf-8')
            f = open(path, mode = 'x')
            f.write(content)
            f.close()
        if flag:
            print('Finish')
```

但这样简单地循环爬取很快就会被豆瓣识别，并要求进行人机验证。若人机验证不通过，则会被直接暂时封IP；并且就算通过了，之后的每次请求也都要验证，这对于这个简单的爬取任务是得不偿失的。

一个思路是每次爬取之后随机等待一段时间，但同一个IP的高频请求仍然会被豆瓣检测到。可以每次被封IP后手动切换IP继续执行爬取，但就算校园网有不同出口IP，手动切换的IP数量是有限的。一个“万无一失”的策略是使用IP代理池；但需要注意的是，就算使用代理池，仍然有一些其它的信息可能暴露，需要多加注意。

此外，由于页面上只显示了部分演职员表的信息，还需要爬取页面 `https://movie.douban.com/subject/:id/celebrities` 上的内容来获取完整的演职员表，爬取方式与爬取电影页面的方式相同，并将其储存在 `movies_staff_data` 中。

电影解析

网页爬取得到的是一个HTML文件，是给用户可视化展示使用的，因此仍需要把它转换成后续程序能够使用的格式化数据，这一部分较复杂。

对于一部电影，网页上记录的有效信息种类如下：电影名，导演，编剧，主演，类型，内容简介，制片国家/地区，语言，又名，演职员表等，除了这些信息以外的其他信息不做处理。

读取到本地储存的电影网页信息后我们创建一个BeautifulSoup对象来访问html内容。读取到的信息最终会被存入一个字典中，索引为上述提到的信息种类。

```
    movie_path = "lab1/movies_data/" + id
    context = open(movie_path, "r", encoding="utf-8")
    soup = BeautifulSoup(context, "html.parser")
```

对于标题项，其内容在标签中。用find()方法可以找到这个标签，若非空则获取其text内容即可得到标题内容。

```
    movie_title_e = soup.find("span", property="v:itemreviewed")
    if movie_title_e:
        movie_title = str(movie_title_e.text)
        contents["title"] = movie_title
```

对于除电影简介和演职员表的剩余信息，其位于 `<div id="info"></div>` 中，每一条信息在 `<span class='pl'></span>` 中。检索到这些标签后分类讨论进行处理。对于类型，其内容在后面 `<span property="v:genre"></span>` 之中，用find_all()方法获取到这些内容后用逗号将其连接方便后面分词与显示。对于其余信息，由于它们所在的标签类型不相同，所以采用由信息种类项用next_sibling()方法获取其

后面的兄弟节点直到找到内容的方法。注意到这些节点中可能包含空格或冒号等无效项，我们需要继续向后寻找。对于剩余的信息种类，我们不对其进行读取。

```
    movie_info = soup.find("div", id="info")
    movie_span = movie_info.find_all("span", {"class": "pl"})
    for i in movie_span:
        temp = str(i.text).replace(":", "")
        if temp == "类型":
            movie_types = movie_info.find_all("span", property="v:genre")
            content = ""
            for movie_type in movie_types:
                content += str(movie_type.text) + ", "
            content = content[: len(content) - 1]
        elif temp in ["导演", "编剧", "主演", "又名", "语言", "制片国家/地区"]:
            tmp = i.next_sibling
            if tmp.text.replace(" ", "") == ":" or tmp.text.replace(" ",
"") == "":
                tmp = tmp.next_sibling
            content = str(tmp.text).replace(" ", "").replace("\n", " ")
        else:
            continue
        contents[temp] = content
```

对于电影简介，其在 `<div class="related-info"><div>` 中的 `<span property="v:summary"><span>` 下，若存在读取即可。注意到内容包含全角空格\u3000，处理时将其删除。

```
    movie_intro = soup.find("div", {"class": "related-info"})
    if movie_intro.find("span", property="v:summary"):
        content = (
            movie_intro.find("span", property="v:summary")
            .text.replace(" ", "")
            .replace("\u3000", "")
        )
        contents["summary"] = content
```

对于演职员表，从movies_staff_data中读取文件。所有的演职员信息在 `<span class="name"><span>` 中，并且演职员负责的工作信息位于其名字的下一个兄弟节点中。将这两个信息作为字典的键与值作为演职员表读取到的信息。

```
    staff_path = "lab1/movies_staff_data/" + id
    content = open(staff_path, "r", encoding="utf-8")
    staff_soup = BeautifulSoup(content, "html.parser")
    staff_names = []
    staff_works = []
    staff_info = staff_soup.find_all("span", {"class": "name"})
    for i in staff_info:
        staff_names.append(str(i.a.text))
        temp = str(i.find_next_sibling().text)
```

```
            staff_works.append(temp)
        staff = dict(zip(staff_names, staff_works))
        contents["author"] = staff
```

以上我们得到了从电影网页中提取到的信息contents，这部分内容可以直接给用户展示但是作为检索的数据仍需做处理。

我们发现对于每一个名字，网页上均会有其中文名与英文名/拼音，为了方便分词我们选择将所有的英文名删除。对于文本中出现的带括号补充信息，发现其大部分为重复信息。最后将一些分隔符改为逗号即可。

```
    for tag in contents:
        temp = contents[tag]
        if tag == "author":
            tmp = dict()
            for j in temp:
                key = re.sub("[a-zA-Z]", "", j).replace(" ",
"").replace("-", "")
                value = re.sub("[a-zA-Z]", "", temp[j]).replace(" ", "")
                tmp[key] = value
            temp = tmp
        else:
            temp = re.sub("[a-zA-Z]", "", temp).replace("-", "")
            temp = re.sub("\\(.*?\\)", "", temp).replace(" ",
"").replace("/", ", ")
        contents_[tag] = temp
```

检索

由于书籍和电影的特征有着较为明显的区别，这里选择书籍和电影分开检索；但实际核心代码使用同一套，仅有外部部分接口的差别。

信息预处理

前面已经得到了格式化的数据。对每一本图书，我们希望将它的作者简介、内容简介等信息分词后，得到所有词汇的列表（而对tag、作者、译者等，应当直接视为一个单词放在列表中）。为了防止由于近义词造成检索失败，应当使用同/近义词表合并词语（需要注意的是，使用近义词表后，匹配的结果就较为模糊了，例如近义词表将中国所有的省份名称视作近义词）。最后的列表还应当根据停用词表过滤。

从文件中构造同/近义词表的代码如下：

```
def get_stopwords():
    stopwords_set = set({"\n", " "})
    with open("lab1/cn_stopwords.txt", "r", encoding="utf-8") as
stopwords_file:
        for line in stopwords_file.readlines():
            stopwords_set.add(line.strip())
    return stopwords_set
```

```
stopwords_set = get_stopwords()


thulac = thulac.thulac(seg_only=True, filt=True)


def get_synonyms():
    synonym_dict = dict()
    with open("lab1/dict_synonym.txt", "r", encoding="utf-8") as
synonym_file:
        for line in synonym_file.readlines():
            ls = line.split()[1:]
            if len(ls) >= 2:
                for i in range(1, len(ls)):
                    synonym_dict[ls[i]] = ls[0]
    return synonym_dict


synonym_dict = get_synonyms()
```

以图书信息处理为例，获取的最终单词列表的代码如下：

```
# split_word.py
def synonym_pivot(word: str):
    if word in synonym_dict:
        return synonym_dict[word]
    else:
        return word


def split_words_book(id: int, method="jieba"):
    data = json.load(open("lab1/book_data/" + str(id) + ".json", "r",
encoding="utf-8"))
    ret = set()
    for s in [data["title"], data["author_intro"], data["summary"]]:
        if method == "jieba":
            for word in jieba.cut_for_search(s):
                ret.add(word)
        elif method == "thulac":
            for word in thulac.cut(s, text=True).split():
                ret.add(word)
    for dc in data["tags"]:
        ret.add(dc["title"])  # use tag as a word
    for key in ["translator", "author"]:
        for name in data[key]:
            ret.add(name)
    ret = set(
        map(
            synonym_pivot,
            ret,
        )
```

```
    )
    return ret - stopwords_set
```

在这里，使用了两个不同分词库：jieba和thulac。

对比这两个分词库的效果，jieba有一个显著的优势：jieba提供了一个专用于搜索引擎的模式，“在精确模式的基础上，对长词再次切分，提高召回率，适合用于搜索引擎分词”。而thulac没有这样的模式，更倾向于把一些长内容视作一个单词，如“顾维钧回忆录”，这样用户搜索“回忆录”时就无法匹配到。因此，在这里选择将jieba作为默认分词工具。

生成倒排索引表

对于文本量上亿的应用，很多系统可能采用词典常驻内存、倒排索引本身在磁盘中的模式。对该任务，一千余条文本信息较少（总共只占用MB级别的内存），且磁盘调度不是算法的核心部分，不妨采用查询时全部常驻内存的方式组织。

这样的话（并且由于该任务中不需要动态插入/删除表项）（为了方便起见），就可以将一个单词的倒排表直接按一个列表的方式存储，而不使用显式的链表。对于跳表指针，采用课件中描述的启发式策略，对一个列表，视作其每隔 $\sqrt{L}$ 有一个跳表指针。倒排索引表整体就以一个键为单词、值为列表的dict类型组织。

要从id列表（已经排序并剔除无效项）中构造倒排表，只需对每个id，将其追加至单词列表中所有单词的列表即可。

```
# inverted_index.py
class InvertedIndex:
    def __init__(self, genre="book") -> None:
        self.table = dict()
        self.genre = genre

    def update_from_id(self, id: int):
        for word in split_words(id, method="jieba", genre=self.genre):
            if word in self.table:
                self.table[word].append(id)
            else:
                self.table[word] = [id]

    def build_from_idlist(self, idlist: str):
        with open(idlist, "r") as idlist_file:
            for id in [int(s.strip()) for s in idlist_file.readlines()]:
                self.update_from_id(id)
        print("Build inverted index succeed!")
```

支持bool查询

这里选择支持的bool查询格式为类似合取范式（只是没有not）的格式，例如 `(A or B) and (C or D or E or ...) and F and ...`。

则需要支持的基本操作即为两个倒排表的and合并与or合并。合并的过程实际上就是在有序序列归并的基础上，增加对（虚拟的）跳表指针的尝试。

```
def and_combine(self, ls1: list, ls2: list):
    n, m = len(ls1), len(ls2)
    b1, b2 = int(sqrt(n)), int(sqrt(m))
    res = []
    i, j = 0, 0
    while i != n and j != m:
        if i % b1 == 0 and i + b1 < n and ls1[i + b1] < ls2[j]:
            i += b1
        if j % b2 == 0 and j + b2 < m and ls2[j + b2] < ls1[i]:
            j += b2
        if ls1[i] == ls2[j]:
            res.append(ls1[i])
            i += 1
            j += 1
        elif ls1[i] < ls2[j]:
            i += 1
        else:
            j += 1
    return res

def or_combine(self, ls1: list, ls2: list):
    n, m = len(ls1), len(ls2)
    b1, b2 = int(sqrt(n)), int(sqrt(m))
    res = []
    i, j = 0, 0
    while i != n and j != m:
        if i % b1 == 0 and i + b1 < n and ls1[i + b1] < ls2[j]:
            res += ls1[i : i + b1]
            i += b1
        if j % b2 == 0 and j + b2 < m and ls2[j + b2] < ls1[i]:
            res += ls2[j : j + b2]
            j += b2
        if ls1[i] == ls2[j]:
            res.append(ls1[i])
            i += 1
            j += 1
        elif ls1[i] < ls2[j]:
            res.append(ls1[i])
            i += 1
        else:
            res.append(ls2[j])
            j += 1
    if i != n:
        res += ls1[i:]
    if j != m:
        res += ls2[j:]
    return res
```

接下来是外层的处理。虽然规定了（本程序中）or的优先级大于and，程序支持用户的输入有至多一层括号或没有括号。解析的实现直接使用split方法。

查询处理的顺序采用了课件中提供的启发式方法，根据or项总数量的估计排序，从小到大执行and。

```
def and_combine_list(self, ls: list):
    if len(ls) == 1:
        return ls[0]
    res = self.and_combine(ls[0], ls[1])
    for i in range(2, len(ls)):
        res = self.and_combine(res, ls[i])
    return res

def or_combine_list(self, ls: list):
    if len(ls) == 1:
        return ls[0]
    res = self.or_combine(ls[0], ls[1])
    for i in range(2, len(ls)):
        res = self.or_combine(res, ls[i])
    return res

def inner_query(self, s: str):
    # requires statements like "(A or B) and (C or D or E or ...) and F
and ..."
    or_items = [w.strip() for w in s.split("and")]
    or_items = [w[1:-1] if w[0] == "(" else w for w in or_items]
    or_items = [[synonym_pivot(o.strip()) for o in w.split("or")] for w in
or_items]
    or_items.sort(key=lambda ls: sum([len(self.query_word(word)) for word
in ls]))
    and_items = [
        self.or_combine_list([self.query_word(word) for word in ls])
        for ls in or_items
    ]
    return self.and_combine_list(and_items)

def query(self, s: str):
    for id in self.inner_query(s):
        print("找到匹配项，id：" + str(id))
        print_info(id, genre=self.genre)
```

最终内容的展现可以展现一些之前提取出的关键信息。例如图书信息的展示代码如下：

```
def print_info_book(id: int):
    data = json.load(open("lab1/book_data/" + str(id) + ".json", "r",
encoding="utf-8"))
    print("书名：" + data["title"])
    print("作者：", end="")
    print(*data["author"], sep="，")
    if len(data["translator"]):
        print("译者：", end="")
        print(*data["translator"], sep="，")
    print("内容简介：" + data["summary"])
```

```
        print("作者简介: " + data["author_intro"])
        print("详情页: " + "https://book.douban.com/subject/" + str(id))
        print()
```

其展示效果如下：

> 找到匹配项，id：1085860 书名：基督山伯爵 作者：[法国] 大仲马 译者：周克希，韩沪麟 内容简介：小说以法国波旁王朝和七月王朝两大时期为背景，描写了一个报恩复仇的故事。法老号大副唐泰斯受船长的临终嘱托，为拿破仑送了一封信，受到 两个对他嫉妒的小人的陷害，被打入死牢，狱友法里亚神甫向他传授了各种知识，还在临终前把一批宝藏的秘密告诉了他。他设法越狱后找到了宝藏，成为巨 富。从此他化名为基督山伯爵，经过精心策划，报答了他的恩人，惩罚了三个一心想置他于死地的仇人。 作者简介：法国19世纪积极浪漫主义作家。其祖父是候爵德·拉·巴那特里，与黑奴结合生下其父，名亚历山大，受洗时用母姓仲马。法国大革命爆发后，亚历 山大·仲马屡建奇功，当上共和政府将军。大仲马终生信守共和政见，一贯反对君主专政，憎恨复辟王朝，不满七月王朝，反对第二帝国。他饱尝种族歧视，心中受到创伤。家庭出身和经历使大仲马形成了反对不平、追求正义的叛逆性格。大仲马自学成才，一生写的各种类型作品达300卷之多，主要以小说和剧作著称于世。大仲马的剧本《享利第三及其宫廷》（1829）比雨果的《欧那尼》还早问世一年。这出浪漫主义戏剧，完全破除了古典主义“三一律”。大仲马小说多达 百部，大都以真实的历史作背景，以主人公的奇遇为内容，情节曲折生动，处处出人意外，堪称历史惊险小说。异乎寻常的理想英雄，急剧发展的故事情节， 紧张的打斗动作，清晰明朗的完整结构，生动有力的语言，灵活机智的对话等构成了大仲马小说的特色。最著名的是《三个火枪手》旧译《三剑客》，（1844 ）、《基督山伯爵》。大仲马被别林斯基称为“一名天才的小说家”，他也是马克思 “最喜欢”的作家之一。 详情页：https://book.douban.com/subject/1085860

存储与压缩

由于之后要压缩索引，因此索引的存储方式应该选用更充分利用空间的二进制存储。

其中，未压缩的存储方式中，每个词项及其列表格式如下：

> 词项长度（2字节）+词项（UTF-8编码）+列表长度（4字节）+id列表（每个数4字节）

直至文件结束。容易写出存储及加载的代码，使用python的struct标准库处理二进制数据：

```
def save(self, path: str):
    # 1.22 MB 1,287,056 bytes
    with open(path, "wb") as file:
        for word, ls in self.table.items():
            bytes = word.encode(encoding="utf-8")
            file.write(struct.pack(">h", len(bytes)))
            file.write(bytes)
            file.write(struct.pack(">i", len(ls)))
            for x in ls:
                file.write(struct.pack(">i", x))
    print("Save to file " + path + " succeed!")
```

```
def load(self, path: str):
    table = dict()
    with open(path, "rb") as file:
        while True:
            name_len_raw = file.read(2)
            if len(name_len_raw) == 0:
                break
            name_len = struct.unpack(">h", name_len_raw)[0]
            bytes = file.read(name_len)
            key = bytes.decode(encoding="utf8")
            ls_len = struct.unpack(">i", file.read(4))[0]
            ls = [struct.unpack(">i", file.read(4))[0] for _ in
range(ls_len)]
            table[key] = ls
    # assert self.table == table  # for testing correctness
    self.table = table
    print("Load from file " + path + " succeed!")
```

要检验算法的正确性，只需在存储后加载，比较前后得到的表是否相同。以图书数据为例，该算法生成的二进制文件大小为1.22MB。

下面考虑压缩。实验文档中提到了“按块存储、前端编码”等压缩方法，这是对正排索引/词典压缩的方法。但作为中文应用，词典有词项长度明显较小、字符集大小大、公共前缀不长等特点，并不方便压缩；而倒排表的大小通常远大于词典大小，因此为了节省空间，在本例中选择压缩倒排表更合适。这里使用“采用间距代替文档ID”的方法（FOR算法），并结合可变长度编码。可变长度编码与解码的代码如下：

```
def varilen_encode(x: int) -> list:
    s = bin(x)[2:]
    ret = bytearray()
    if len(s) % 7 != 0:
        ret.append(int(s[0 : len(s) % 7], base=2))
    for i in range(len(s) % 7, len(s), 7):
        ret.append(int(s[i : i + 7], base=2))
    ret[-1] += 1 << 7
    return ret


def varilen_decode(ls: list) -> int:
    ret = 0
    for x in ls:
        if x >= 1 << 7:
            ret = (ret << 7) + x - (1 << 7)
        else:
            ret = (ret << 7) + x
    return ret
```

有了上面两个函数，就可以方便地修改存储/加载函数为压缩后的版本：

```
def save_compressed(self, path):
    # 951 KB 974,417 bytes
    with open(path, "wb") as file:
        for word, ls in self.table.items():
            bytes = word.encode(encoding="utf-8")
            file.write(struct.pack(">h", len(bytes)))
            file.write(bytes)
            file.write(struct.pack(">i", len(ls)))
            last = 0
            for x in ls:
                file.write(varilen_encode(x - last))
                last = x
    print("Save(compressed) to file " + path + " succeed!")
```

```
def load_compressed(self, path: str):
    table = dict()
    with open(path, "rb") as file:
        while True:
            name_len_raw = file.read(2)
            if len(name_len_raw) == 0:
                break
            name_len = struct.unpack(">h", name_len_raw)[0]
            bytes = file.read(name_len)
            key = bytes.decode(encoding="utf8")
            ls_len = struct.unpack(">i", file.read(4))[0]
            last = 0
            ls = []
            while ls_len > 0:
                bytes = bytearray()
                while True:
                    bytes.append(file.read(1)[0])
                    if bytes[-1] >= 1 << 7:
                        break
                x = last + varilen_decode(bytes)
                ls.append(x)
                last = x
                ls_len -= 1
            table[key] = ls
    # assert self.table == table  # for testing correctness
    self.table = table
    print("Load(compressed) from file " + path + " succeed!")
```

压缩后的图书倒排索引大小为951KB，为原来的75.7%，有明显的效果。

第一阶段测试

第一阶段结果的主代码位于 `inverted_index.py` 中。对倒排索引各功能的测试代码如下：

```
if __name__ == "__main__":
    l = InvertedIndex(genre="book")
    l.build_from_idlist("lab1/Book_id.csv")
    l.save("lab1/Book_inverted.bin")
    l.load("lab1/Book_inverted.bin")
    l.save_compressed("lab1/Book_inverted_compressed.bin")
    l.load_compressed("lab1/Book_inverted_compressed.bin")

    # top39 基督山伯爵
    # top139 阿勒泰的角落
    l.query("(法国 or 新疆 or 北京) and (大仲马 or 李娟 or 老舍) and (复仇 or 日常 or 骆驼)")

    l = InvertedIndex(genre="movie")
    l.build_from_idlist("lab1/Movie_id.csv")
    l.save("lab1/Movie_inverted.bin")
    l.load("lab1/Movie_inverted.bin")
    l.save_compressed("lab1/Movie_inverted_compressed.bin")
    l.load_compressed("lab1/Movie_inverted_compressed.bin")

    # top37 哈尔的移动城堡
    # top137 玩具总动员3
    l.query("(玩具 or 城堡) and 妹妹")
```

其中，两行query是对查询效果的测试。根据top39《基督山伯爵》、top139《阿勒泰的角落》，以及随机选取的《骆驼祥子》，构造出查询如下：

> (法国 or 新疆 or 北京) and (大仲马 or 李娟 or 老舍) and (复仇 or 日常 or 骆驼)

查询得到了期望的书籍；由于查询条件较严格，没有更多的匹配项。

而对于电影查询，根据top37《哈尔的移动城堡》、top137《玩具总动员3》构造出查询"(玩具 or 城堡) and 妹妹"；由于筛选条件较宽松，除期望的两个结果外，还返回了符合条件的《蓝精灵 第一季》等。

第二阶段

MF（Matrix Factorization）模型

模型说明

MF (矩阵分解) 是将一个矩阵分解为两个或多个矩阵，使得分解的矩阵能够通过相乘得到原始矩阵的行为。此处它根据 user-item 评分矩阵来预测用户对未打分物品的评分行为。

用 U 表示用户的集合， D 表示物品的集合， R 表示用户评分矩阵，可以将用户和物品映射到一个 K 维的潜在特征空间。通过挖掘用户的潜在特征矩阵 P (|U| x K维) 和物品潜在特征矩阵 Q (|D| x K维) 来估计评分。即通过下式来得到评分矩阵：

$$R \approx P \times Q^T = \hat{R}$$

可以计算出用户 user 对物品 item 的预测评分

$$\hat{r_{ui}} = p_u q_i^T$$

误差

我们将误差拆解成 4 个部分: 平均打分 mean，物品偏差 $item_b ias$，用户偏差 $user_b ias$，估计评分 $p^T q$。

$item_b ias$ 表示一个 item 在同题材下评分较于平均分的差值，$user_b ias$ 表示一位 user 评分较于大众评分的差值。

预测与优化目标

根据如上分析我们得出 user 对于 item 的评分可以表示为：

$$\hat{r_{ui}} = mean + bias_u + p_u q_i^T$$

则模型需要优化的目标即为

$$\min_{p,q,bias} \sum_{(user,item)\in K} (r_{ui} - mean - bias_i - bias_u - p_u q_i^T)^2$$

具体实现

```
class MF(nn.Module):
    def __init__(self, user_num, item_num, mean, size, dropout):
        super(MF, self).__init__()
        self.user_ebds = nn.Embedding(user_num, size)
        self.item_ebds = nn.Embedding(item_num, size)
        self.user_bias = nn.Embedding(user_num, 1)
        self.item_bias = nn.Embedding(item_num, 1)
        self.mean = nn.Parameter(torch.FloatTensor([mean]), False)
        self.dropout = nn.Dropout(dropout)

    def forward(self, user_id, item_id):
        user_ebd = self.user_ebds(user_id)
        item_ebd = self.item_ebds(item_id)
        user_b = self.user_bias(user_id).squeeze()
        item_b = self.item_bias(item_id).squeeze()
        return self.dropout((user_ebd * item_ebd).sum(1) + user_b + item_b
+ self.mean)
```

数据处理

首先对电影、书籍和用户编号进行正则化处理，将给电影评分的用户编号为0-1023，给书籍评分的用户编号为0-4419，电影和书籍编号为0-1200。 将新数据存入 `book_score_clean.csv` 和 `movie_score_clean.csv` 中作为模型的原始数据。

```
    with open(data_path, mode = 'r') as file:
        data = csv.reader(file)
        info = next(data)
```

```
        data_clean.append(info)
        user_id = 0
        item_id = 0
        for line in data:
            temp = line
            if line[0] not in user_idx.keys():
                user_idx[line[0]] = user_id
                user_id += 1
            if line[1] not in item_idx.keys():
                item_idx[line[1]] = item_id
                item_id += 1
            temp[0] = user_idx[temp[0]]
            temp[1] = item_idx[temp[1]]
            data_clean.append(temp)
```

随后将数据按时间戳排序，按照 5 : 5 的比例划分为训练集和测试集。

```
    train = data.loc[: int(0.5 * n) - 1]
    test = data.loc[int(0.5 * n): ]
    test_dic = [dict() for i in range(user_num)]
    for user in range(user_num):
        items = test.loc[test['User'] == user]
        for i, item in items.iterrows():
            test_dic[user][int(item[type])] = int(item['Rate'])
```

结果评估

用 ndcg 评估结果，具体步骤如下：

1. 对于测试集中的每一个 user ，通过训练得到的模型预测出其在测试集中 item 的评分，按评分排序得到 predict ，将其与真实得分一起算出 DCG。

```
    item_key = list(item_dic.keys())
    user = torch.full((len(item_key), ), user_id, dtype = torch.int64)
    item = torch.tensor(item_key, dtype = torch.int64)
    predict = model(user, item).detach().cpu().numpy()[: , np.newaxis]
    temp = np.array(list(item_dic.values()))[: , np.newaxis]
    ranks = np.concatenate([predict, temp], axis = 1).tolist()
    ranks.sort(reverse = True)
    for i, (tmp, rank) in enumerate(ranks):
        DCG += (2**rank - 1) / log2(i + 2)
```

2. 将该 user 在测试集中 item 按评分由高到低排序得到 item_value ，用其与真实排序计算出 IDCG。

```
    item_value = list(item_dic.values())
    item_value.sort(reverse = True)
    for i, rank in enumerate(item_value):
```

```
            IDCG += (2**rank - 1) / log2(i + 2)
        if IDCG:
            NDCG += DCG / IDCG
            n += 1
```

3. DCG / IDCG = NDCG ，对所有用户求平均即得最后模型的评分。

结果分析

训练 20 次模型，观察它们的 NDCG ，结果如下：

Book:

```
Book
100%|| 156/156 [00:06<00:00, 23.40it/s]
Epoch 0: test ndcg is 0.7341329280782516
100%|| 156/156 [00:06<00:00, 23.63it/s]
Epoch 1: test ndcg is 0.7357217721045806
100%|| 156/156 [00:06<00:00, 23.08it/s]
Epoch 2: test ndcg is 0.7358216367753283
100%|| 156/156 [00:06<00:00, 23.59it/s]
Epoch 3: test ndcg is 0.736123699969698
100%|| 156/156 [00:06<00:00, 23.19it/s]
Epoch 4: test ndcg is 0.734830011004753
100%|| 156/156 [00:06<00:00, 23.52it/s]
Epoch 5: test ndcg is 0.7323810449471766
100%|| 156/156 [00:06<00:00, 23.01it/s]
Epoch 6: test ndcg is 0.7327593710128735
100%|| 156/156 [00:06<00:00, 23.14it/s]
Epoch 7: test ndcg is 0.7327925502979499
100%|| 156/156 [00:06<00:00, 23.11it/s]
Epoch 8: test ndcg is 0.7316980183935199
100%|| 156/156 [00:06<00:00, 23.33it/s]
Epoch 9: test ndcg is 0.7298287044016584
100%|| 156/156 [00:06<00:00, 23.14it/s]
Epoch 10: test ndcg is 0.7325734863714837
100%|| 156/156 [00:06<00:00, 23.92it/s]
Epoch 11: test ndcg is 0.7325723401148738
100%|| 156/156 [00:06<00:00, 23.25it/s]
Epoch 12: test ndcg is 0.7316834010961686
100%|| 156/156 [00:06<00:00, 23.37it/s]
Epoch 13: test ndcg is 0.7336704003976394
100%|| 156/156 [00:06<00:00, 23.05it/s]
Epoch 14: test ndcg is 0.7339939648151245
100%|| 156/156 [00:06<00:00, 23.19it/s]
Epoch 15: test ndcg is 0.7325440811049246
100%|| 156/156 [00:06<00:00, 23.55it/s]
Epoch 16: test ndcg is 0.7304945829988779
100%|| 156/156 [00:06<00:00, 23.11it/s]
Epoch 17: test ndcg is 0.7305101319546059
100%|| 156/156 [00:06<00:00, 23.22it/s]
Epoch 18: test ndcg is 0.7307952398738194
100%|| 156/156 [00:06<00:00, 22.90it/s]
Epoch 19: test ndcg is 0.7321298677271064
```

平均 NDCG 约在0.733左右

Movie:

```
Movie
100%|| 175/175 [00:07<00:00, 24.17it/s]
Epoch 0: test ndcg is 0.803310128617548
100%|| 175/175 [00:07<00:00, 22.62it/s]
Epoch 1: test ndcg is 0.807847060739236
100%|| 175/175 [00:07<00:00, 22.79it/s]
Epoch 2: test ndcg is 0.8063041520614459
100%|| 175/175 [00:07<00:00, 22.71it/s]
Epoch 3: test ndcg is 0.8046655139288075
100%|| 175/175 [00:07<00:00, 22.52it/s]
Epoch 4: test ndcg is 0.8014712786492059
100%|| 175/175 [00:07<00:00, 22.73it/s]
Epoch 5: test ndcg is 0.800537878893494
100%|| 175/175 [00:07<00:00, 22.52it/s]
Epoch 6: test ndcg is 0.8000789476099143
100%|| 175/175 [00:07<00:00, 22.91it/s]
Epoch 7: test ndcg is 0.8006993294623037
100%|| 175/175 [00:07<00:00, 23.00it/s]
Epoch 8: test ndcg is 0.8004073801628555
100%|| 175/175 [00:07<00:00, 22.71it/s]
Epoch 9: test ndcg is 0.8003632030249394
100%|| 175/175 [00:07<00:00, 22.82it/s]
Epoch 10: test ndcg is 0.8006127460893495
100%|| 175/175 [00:07<00:00, 22.69it/s]
Epoch 11: test ndcg is 0.8006935979958898
100%|| 175/175 [00:07<00:00, 22.64it/s]
Epoch 12: test ndcg is 0.8011147234768458
100%|| 175/175 [00:07<00:00, 22.78it/s]
Epoch 13: test ndcg is 0.799165444720161
100%|| 175/175 [00:07<00:00, 22.64it/s]
Epoch 14: test ndcg is 0.7990768237779127
100%|| 175/175 [00:07<00:00, 22.78it/s]
Epoch 15: test ndcg is 0.7987384042598075
100%|| 175/175 [00:07<00:00, 22.68it/s]
Epoch 16: test ndcg is 0.7990864434963758
100%|| 175/175 [00:07<00:00, 22.63it/s]
Epoch 17: test ndcg is 0.7991004532203632
100%|| 175/175 [00:07<00:00, 22.80it/s]
Epoch 18: test ndcg is 0.7986153499470948
100%|| 175/175 [00:07<00:00, 23.51it/s]
Epoch 19: test ndcg is 0.7983075347864007
```

平均 NDCG 约在0.801左右

随后我们取一次训练出的模型，将训练集中预测出的排序与实际排序整理进一个表 `data/:item_rank.csv` 中。通过观察数据集，我们发现大部分的 item 排序相对位置都基本正确。

提交文件目录

```
├──stage 1
│    │   crawl.py             # 书籍信息的爬取
│    │   erase.py             # 爬取到无效信息的删除
│    │   inverted_index.py    # 生成倒排索引表
│    │   movie_ex.py          # 电影信息的提取
│    │   movie_fetch.py       # 电影信息的爬取
│    │   print_info.py        # 信息展示
│    │   split_words.py       # 分词
│    │
│    └──data
│        │   Book_id.csv                       # 需要爬取的书籍id
│        │   Book_tag.csv                      # 书籍标签
│        │   Book_inverted.bin                 # 书籍倒排表的一个例子
│        │   Book_inverted_compressed.bin      # 压缩后的书籍倒排表
│        │   Movie_id.csv                      # 需要爬取的电影id
│        │   Movie_tag.csv                     # 电影标签
│        │   Movie_inverted.bin                # 电影倒排表的一个例子
│        │   Movie_inverted_compressed.bin     # 压缩后的电影倒排表
│        │   cn_stopwords.txt                  # 停用词表
│        │   dict_synonym.txt                  # 同义词表
│        │
│        ├──book_data
│        │       ··· # 爬取到的书籍信息
│        │
│        ├──movies_data
│        │       ··· # 爬取到的电影信息
│        │
│        └──movies_staff_data
│                ··· # 爬取到的电影演职员表信息
│
└──stage 2
    │   Data_split.py    # 数据处理
    │   Evaluate.py      # 建立倒排表
    │   Model.py         # 用自然语言来搜索电影
    │   Train.py         # 分词
    │
    └──data
            book_score.csv           # 书籍评分原始数据
            book_score_clean.csv     # 处理后的书籍数据
            book_rank.csv            # 模型对书籍的预测
            movie_score.csv          # 电影评分原始数据
            movie_score_clean.csv    # 处理后的电影数据
            movie_rank.csv           # 模型对电影的预测
```