

实验一 实验报告

组长：龙奕均 PB22111659

组员：林琦皓 PB22051003， 杨柄权 PB22111686

实验仓库：<https://github.com/lqh1106/webinfo.git>

实验一 实验报告

第 1 阶段 豆瓣数据的检索

实验任务

实验内容

实验方法

数据预处理

建立倒排表

实现布尔查询

索引压缩

关键代码说明

get_tags.py

list_generate.py

Qbool.py

Qbool_optimized.py

list_storage.py

结果分析

预处理效果分析

分词结果对比

同义词替换效果分析

删除停用词效果分析

布尔查询效率分析

索引压缩效果对比

存储空间对比

检索效率对比

第 2 阶段 豆瓣数据的个性化检索与推荐

实验任务

实验内容

实验方法

基本思想

基础实验

选做实验

关键代码说明

项目相似程度的度量

计算 tfidf

相似度计算

历史数据获取

社交网络遍历

获取社交网络

计算用户的朋友的评分信息的倒排索引

获取评分项目的特征向量

K 近邻分析

加权平均

线性回归

性能测试方法

MSG

NDCG

结果分析

第 1 阶段 豆瓣数据的检索

实验任务

根据给定的豆瓣 Movie&Book 的 tag 信息，实现电影和书籍的检索（可以合在一起做或者分别做一遍）。对于给定的查询，通过分词、构建索引、索引优化等流程，实现面向给定查询条件的精确索引。

实验内容

1. 对给定的电影和书籍数据进行预处理，将文本表征为关键词集合；
2. 在经过预处理的数据集上建立倒排索引表 S ，并以合适的方式存储生成的倒排索引文件；
3. 对于给定的包含任意组合（如括号）的布尔查询，使其能够支持复杂的布尔查询操作。返回符合查询规则的电影或/和书籍集合，并以合适的方式展现给用户；
4. 任选两种课程中介绍过的索引压缩方法加以实现，并比较压缩后的索引在存储空间和检索效率上与原索引的区别。

实验方法

数据预处理

1. 使用 openccc 库，将分词结果中的繁体字替换为对应的简体字，以便于分词；
2. 使用三种方式进行分词，分别为双向最大匹配分词，结巴分词和 pkuseg，以便于对分词结果进行对比；
3. 使用从词林获取的手工标注的同/近义词表，对语义相似的词语进行合并，替换为近义词表的对应行的第一项；
4. 使用百度的停用词表，将停用词从分词结果中去除；
5. 完成以上步骤后，将预处理的数据存储在 csv 文件中。

建立倒排表

1. 使用预处理后的数据建立数据结构为字典的单层倒排索引表；
2. 将倒排索引表存储在 json 文件中，以便读取。

实现布尔查询

不进行优化的复杂布尔查询实现方法如下：

1. 使用栈来分析复杂的布尔查询的查询条件；
2. 识别到查询条件中的标签关键字时，先将该关键字替换为近义词表的对应行的第一项，再在倒排索引表中进行查询，并将查询结果储存在栈上；
3. 运算符的处理：对于 AND 运算符，利用集合的交集实现；对于 OR 运算符，利用集合的并集实现；对于 NOT 运算符，利用集合的补集实现；

利用真值表进行优化的复杂布尔查询实现方法如下：

1. 提取出复杂布尔查询的所有布尔变量，生成其对应的索引字典，以缩小查询范围；
2. 对该布尔查询，使用真值表遍历所有的真值组合；
3. 对某一真值组合内部，输出集合为所有值为真的布尔变量对应的索引字典中的集合的交集与所有值为假的布尔变量对应的索引字典中的集合的并集相减得到的集合；
4. 查询结果为所有真值集合的并集。

索引压缩

实现的两种索引压缩的方法如下：

1. 第一步压缩：使用间距代替文档 ID；
2. 进一步压缩：对间距进行可变长度编码。

将压缩完的倒排索引表打包存储在 .pkl 文件中，通过对比文件大小来比较存储空间；由于在布尔查询初始化时，已经将压缩后的倒排索引表解压至内存中了，故索引压缩不会影响检索效率，只会影响初始化时的解压速度。

关键代码说明

get_tags.py

实验任务中数据预处理的部分在 lab1-1/get_tags.py 中实现，以下是关键代码说明：

```
convert_text(text, conversion_type='t2s')
```

将文本从简体中文转换为繁体中文，或反之。

- **参数:**
 - `text`: 要转换的文本。
 - `conversion_type`: 转换类型，默认值为 't2s'（繁体到简体）。
- **返回:** 转换后的文本。
- **异常:** 如果提供的转换类型不支持则抛出 `ValueError`。

```
def convert_text(text, conversion_type='s2t'):
    if conversion_type == 's2t':
        return converter_s2t.convert(text)
    elif conversion_type == 't2s':
        return converter_t2s.convert(text)
    else:
        raise ValueError(
            "Unsupported conversion type. Use 's2t' for Simplified to Traditional or 't2s' for Traditional to Simplified.")
```

```
load_synonym_dict(file_path)
```

加载同义词词典，并返回一个字典。

- **参数:**
 - `file_path`: 同义词文件的路径。
- **返回:** 包含同义词关系的字典，词汇映射到中心词。

```
def load_synonym_dict(file_path):
    synonym_dict = {}
    code = ''
    with open(file_path, 'r', encoding='utf-8') as file:
        for line in file:
            # 使用正则表达式匹配 '=', '#' 或 '@' 作为分隔符
            newcode, synonyms = re.split(r'[#@]\s', line.strip(), maxsplit=1)
            synonym_list = synonyms.split(' ')
            if synonym_list: # 确保有近义词
                if newcode[:7] != code[:7]:
```

```
        code = newcode
        center_word = synonym_list[0] # 第一个词作为中心词
    for word in synonym_list:
        synonym_dict[word] = center_word
    return synonym_dict
```

```
replace_with_center_word(words, synonym_dict)
```

用同义词词典中的中心词替换输入词汇。

- **参数:**
 - `words`: 要处理的词汇列表。
 - `synonym_dict`: 同义词字典。
- **返回:** 替换后的词汇列表。

```
def replace_with_center_word(words, synonym_dict):
    new_words = [synonym_dict.get(word, word) for word in words]
    return new_words
```

```
save_data(data, file_path)
```

将数据保存为 CSV 文件。

- **参数:**
 - `data`: 要保存的数据，通常为字典格式。
 - `file_path`: 输出文件的路径。

```
def save_data(data, file_path):
    df = pd.DataFrame(data)
    # Save as CSV
    df.to_csv(file_path, index=False, encoding='utf-8')
```

```
load_stopwords(filepath)
```

加载停用词列表并返回一个集合。

- **参数:**
 - `filepath`: 停用词文件的路径。
- **返回:** 停用词的集合。

```
def load_stopwords(filepath):
    with open(filepath, 'r', encoding='utf-8') as f:
        stopwords = f.read().splitlines()
    return set(stopwords)
```

```
is_numeric_or_symbol(word)
```

检查输入的单词是否为纯数字或符号。

- **参数:**
 - `word`: 要检查的单词。

- **返回:** 如果是纯数字或纯符号字符串则返回 `True` , 否则返回 `False`。

```
def is_numeric_or_symbol(word):  
    # 匹配纯数字或纯符号的字符串  
    return bool(re.fullmatch(r'[\d]+|[\^\w\s]+', word))
```

```
get_tags(original_data, original_data_type, stopwords, synonym_dict, cut_type,  
file_path, pkuseg)
```

从原始数据中提取标签, 进行分词、同义词替换和停用词过滤, 并将结果保存。

- **参数:**
 - `original_data`: 原始数据, 包含标签。
 - `original_data_type`: 数据类型 (Movie 或 Book) 。
 - `stopwords`: 停用词集合。
 - `synonym_dict`: 同义词字典。
 - `cut_type`: 分词类型, 可以是 'jieba' 或 'pkuseg'。
 - `file_path`: 输出文件的路径。
 - `pkuseg`: 用于分词的 PKU 分词对象。
- **返回:** 无, 结果保存到文件中。

```
def get_tags(original_data, original_data_type, stopwords, synonym_dict,  
cut_type, file_path, pkuseg):  
    data = {original_data_type: [], "Tags": []}  
    for number, tags in zip(original_data[original_data_type],  
original_data['Tags']):  
        tags = tags.strip("{}")  
        tags_list = [item.strip().strip(",")  
                      for item in tags.split(",")]  
        words = []  
        for tag in tags_list:  
            simplified_tag = convert_text(tag, 't2s') #繁体转简体  
            if cut_type == 'jieba':  
                cut_words = [word for word in jieba.lcut(  
simplified_tag) if not is_numeric_or_symbol(word)] #jieba分  
词, 并删除纯数字或符号串  
            elif cut_type == 'pkuseg':  
                cut_words = [word for word in pkuseg.cut(  
simplified_tag) if not is_numeric_or_symbol(word)] #pkuseg分  
词, 并删除纯数字或符号串  
            a = replace_with_center_word(cut_words, synonym_dict) # 同义词替换为中  
心词  
            a = [word for word in a if word not in stopwords] #删除停用词  
            words += a  
            data[original_data_type].append(number)  
            data['Tags'].append(words)  
        save_data(data, file_path)
```

```
is_english(text)
```

判断输入的文本是否为英文。

- **参数:**

- `text`: 要检查的文本。
- **返回**: 如果文本全为英文字符, 则返回 `True`, 否则返回 `False`。

```
def is_english(text):  
    # 判断是否为英文  
    return bool(re.match(r'^[a-zA-Z]+$', text))
```

```
bidirectional_maximum_matching(text, dictionary)
```

实现双向最大匹配算法, 对文本进行分词。

- **参数**:
 - `text`: 要分词的文本。
 - `dictionary`: 用于分词的词典。
- **返回**: 分词后的结果列表。

```
def bidirectional_maximum_matching(text, dictionary):  
    # 正向最大匹配  
    def forward_match(text):  
        words = []  
        while text:  
            for i in range(len(text), 0, -1):  
                # 将英文单词整个保存, 不分词  
                if is_english(text[:i]):  
                    words.append(text[:i])  
                    text = text[i:]  
                    break  
                if text[:i] in dictionary:  
                    words.append(text[:i])  
                    text = text[i:]  
                    break  
            else:  
                words.append(text[0])  
                text = text[1:]  
        return words  
  
    # 反向最大匹配  
    def backward_match(text):  
        words = []  
        while text:  
            for i in range(len(text), 0, -1):  
                # 将英文单词整个保存, 不分词  
                if is_english(text[-i:]):  
                    words.append(text[-i:])  
                    text = text[:-i]  
                    break  
                if text[-i:] in dictionary:  
                    words.append(text[-i:])  
                    text = text[:-i]  
                    break  
            else:  
                words.append(text[-1])  
                text = text[:-1]  
        return words[::-1]
```

```

forward_words = forward_match(text)
backward_words = backward_match(text)

# 选择较短的分词结果
if len(forward_words) >= len(backward_words):
    return backward_words
return forward_words

```

```

get_tags_maxmatch(original_data, original_data_type, dictionary, stopwords,
synonym_dict, file_path)

```

使用双向最大匹配分词法从原始数据中提取标签，并保存结果。

- **参数:**
 - `original_data`: 原始数据，包含标签。
 - `original_data_type`: 数据类型 (Movie 或 Book) 。
 - `dictionary`: 用于分词的词典。
 - `stopwords`: 停用词集合。
 - `synonym_dict`: 同义词字典。
 - `file_path`: 输出文件的路径。
- **返回:** 无，结果保存到文件中。

```

def get_tags_maxmatch(original_data, original_data_type, dictionary, stopwords,
synonym_dict, file_path):
    data = {original_data_type: [], "Tags": []}
    for number, tags in zip(original_data[original_data_type],
original_data['Tags']):
        tags = tags.strip("{}")
        tags_list = [item.strip().strip(",'")
                      for item in tags.split(",")]
        words = []
        for tag in tags_list:
            simplified_tag = convert_text(tag, 't2s') #繁体转简体
            cut_words = [
                word for word in bidirectional_maximum_matching(simplified_tag,
dictionary) if not
                is_numeric_or_symbol(word)]
            a = replace_with_center_word(cut_words, synonym_dict)# 同义词替换为中心
词
            a = [word for word in a if word not in stopwords]#删除停用词
            words += a
            data[original_data_type].append(number)
            data['Tags'].append(words)
        save_data(data, file_path)

```

list_generate.py

实验任务中建立倒排表的部分在 lab1-1/list_generate.py 中实现，以下是关键代码说明：

```

generate_inverted_index(file_path, file_type, output_path)

```

从给定的 CSV 文件生成倒排索引，并将其保存为 JSON 文件。

- **参数:**
 - `file_path`: 输入的预处理后的 csv 文件。

- `file_type`: 数据类型 (Movie 或 Book) 。
- `file_path`: 输出文件的路径。
- **返回**: 无, 结果保存到文件中。

```
def generate_inverted_index(file_path, file_type, output_path):
    """
    从给定的CSV文件生成倒排索引，并将其保存为JSON文件。
    """
    # 读取CSV文件
    data = pd.read_csv(file_path)

    # 初始化倒排索引字典
    inverted_index = {}

    # 遍历每一行数据
    for index, row in data.iterrows():
        book_id = row[file_type]
        # 将字符串形式的列表转换为真实的Python列表
        tags = ast.literal_eval(row['Tags'])

        # 遍历每个标签
        for tag in set(tags): # 使用set去重，避免重复
            if tag not in inverted_index:
                inverted_index[tag] = []
            inverted_index[tag].append(book_id)

    # 对倒排索引表中的文档ID列表进行排序
    for tag in inverted_index:
        inverted_index[tag].sort()
        # 储存频率和文档ID列表
        inverted_index[tag] = [len(inverted_index[tag]), inverted_index[tag]]

    # 将倒排表按tag排序
    inverted_index = dict(sorted(inverted_index.items()))

    # 将倒排索引字典写入到一个JSON文件中
    with open(output_path, 'w', encoding='utf-8') as f:
        json.dump(inverted_index, f, ensure_ascii=False, indent=4)

    print(f"倒排索引表已生成并存储到 '{output_path}' 文件中。")
```

Qbool.py

实验任务中基本的布尔查询的部分在 `lab1-1/Qbool.py` 中实现，以下是关键代码说明：

```
load_inverted_index(file_path)
```

从指定路径加载倒排索引文件，并返回解析后的 JSON 对象。

- **参数**:
 - `file_path`: 倒排索引文件的路径。
- **返回**:
 - 解析后的倒排索引数据。


```
def load_inverted_index(file_path):
    with open(file_path, 'r', encoding='utf-8') as f:
        return json.load(f)
```

```
load_synonym_dict(file_path)
```

加载同义词词典，将其解析为字典形式。

- **参数:**
 - `file_path`: 同义词文件的路径。
- **返回:**
 - 包含同义词关系的字典，词汇映射到中心词。

```
def load_synonym_dict(file_path):
    synonym_dict = {}
    code = ''
    with open(file_path, 'r', encoding='utf-8') as file:
        for line in file:
            # 使用正则表达式匹配 '=', '#' 或 '@' 作为分隔符
            newcode, synonyms = re.split(r'[=#@]\s', line.strip(), maxsplit=1)
            synonym_list = synonyms.split(' ')
            if synonym_list: # 确保有近义词
                if newcode[:7] != code[:7]:
                    code = newcode
                    center_word = synonym_list[0] # 第一个词作为中心词
                for word in synonym_list:
                    synonym_dict[word] = center_word
    return synonym_dict
```

```
replace_with_center_word(word, synonym_dict)
```

用同义词词典中的中心词替换输入词汇。

- **参数:**
 - `word`: 要处理的词汇。
 - `synonym_dict`: 同义词字典。
- **返回:**
 - 替换后的词汇。

```
def replace_with_center_word(word, synonym_dict):
    new_word = synonym_dict.get(word, word)
    return new_word
```

```
boolean_query_basic(inverted_index, query, synonyms_dict)
```

执行基本的布尔查询，处理操作符并返回符合条件的结果集。

- **参数:**
 - `inverted_index`: 倒排索引数据。
 - `query`: 用户输入的查询字符串。
 - `synonyms_dict`: 同义词字典。

- 返回:
 - 满足查询条件的书籍 ID 集合。

```
def boolean_query_basic(inverted_index, query, synonyms_dict):
    # 去除多余的空格
    query = query.strip()

    # 定义操作符
    operators = {'AND', 'OR', 'NOT'}

    # 定义运算符的优先级
    precedence = {'AND': 2, 'OR': 1, 'NOT': 3}

    # 用于存储操作数和运算符的栈
    output_stack = []
    operator_stack = []

    # 格式化括号
    tokens = query.replace('(', ' ( ').replace(')', ' ) ').split()

    # 处理查询
    for token in tokens:
        if token not in operators and token != '(' and token != ')':
            # 不在运算符中的是标签，获取其对应ID
            token = replace_with_center_word(token, synonyms_dict)
            data = inverted_index.get(token, [0])
            if len(data) > 1:
                output_stack.append((len(data[1]), set(data[1]))) # 存储频率和集合
            else:
                output_stack.append((0, set())) # 如果没有ID，加入频率为0的空集合
        elif token == '(':
            operator_stack.append(token)
        elif token == ')':
            while operator_stack and operator_stack[-1] != '(':
                operator = operator_stack.pop()
                if operator == 'NOT':
                    operand = output_stack.pop()[1]
                    all_ids = set()
                    for value in inverted_index.values():
                        if len(value) > 1:
                            all_ids.update(set(value[1]))
                    output_stack.append((len(all_ids - operand), all_ids -
operand)) # 进行集合的补集运算
                else:
                    right = output_stack.pop()
                    left = output_stack.pop()
                    if operator == 'AND':
                        result_set = left[1].intersection(right[1])
                        output_stack.append((len(result_set), result_set)) # 取
交集
                    elif operator == 'OR':
                        result_set = left[1].union(right[1])
                        output_stack.append((len(result_set), result_set)) # 取
并集
                    operator_stack.pop() # 弹出 '('
            else:
                while (operator_stack and operator_stack[-1] != '(' and
```

```

        precedence[token] <= precedence[operator_stack[-1]]):
    operator = operator_stack.pop()
    if operator == 'NOT':
        operand = output_stack.pop()[1]
        all_ids = set()
        for value in inverted_index.values():
            if len(value) > 1:
                all_ids.update(set(value[1]))
        output_stack.append((len(all_ids - operand), all_ids -
operand)) # 进行集合的补集运算
    else:
        right = output_stack.pop()
        left = output_stack.pop()
        if operator == 'AND':
            result_set = left[1].intersection(right[1])
            output_stack.append((len(result_set), result_set)) # 取
交集
        elif operator == 'OR':
            result_set = left[1].union(right[1])
            output_stack.append((len(result_set), result_set)) # 取
并集

    operator_stack.append(token)

# 处理剩余的运算符
while operator_stack:
    operator = operator_stack.pop()
    if operator == 'NOT':
        operand = output_stack.pop()[1]
        all_ids = set()
        for id_list in inverted_index.values():
            all_ids.update(set(id_list))
        output_stack.append((len(all_ids - operand), all_ids - operand)) # 进
行集合的补集运算
    else:
        right = output_stack.pop()
        left = output_stack.pop()
        if operator == 'AND':
            result_set = left[1].intersection(right[1])
            output_stack.append((len(result_set), result_set)) # 取交集
        elif operator == 'OR':
            result_set = left[1].union(right[1])
            output_stack.append((len(result_set), result_set)) # 取并集

# 返回最终结果
return output_stack[0][1] if output_stack else set()

```

Qbool_optimized.py

实验任务中优化后的布尔查询的部分在 lab1-1/Qbool_optimized.py 中实现，以下是关键代码说明（与 Qbool.py 中重复的函数不另作说明）：

```
extract_variables(query_string, synonym_dict)
```

从查询字符串中提取布尔变量并替换为同义词。

- 参数:

- query_string: 要提取变量的布尔查询表达式字符串。

- `synonym_dict`: 用于替换变量的同义词字典。
- **返回**: 一个元组, 包含两个元素:
 - `found_variables`: 查询中所有布尔变量的列表。
 - `new_string`: 替换同义词后的查询字符串。

```
def extract_variables(query_string, synonym_dict):  
    # 使用正则表达式提取变量, 并替换同义词  
    pattern = r'\b(?:AND|OR|NOT)(\w+)\b' # 匹配字母组成的单词  
    all_variables = re.findall(pattern, query_string)  
    found_variables = replace_with_center_word(all_variables, synonym_dict)  
    new_string = query_string  
    # 同时返回替换同义词后的字符串, 如何返回?  
    for var in all_variables:  
        new_string = new_string.replace(var, synonym_dict.get(var, var))  
  
    return found_variables, new_string
```

`generate_index_dict(inverted_index, variables)`

生成给定布尔变量的索引字典。

- **参数**:
 - `inverted_index`: 存储反向索引数据的字典。
 - `variables`: 要生成索引字典的变量列表。
- **返回**: 索引字典, 包含每个变量及其对应文档索引的集合。

```
def generate_index_dict(inverted_index, variables):  
    # 生成索引字典  
    index_dict = {}  
    for var in variables:  
        if var in inverted_index:  
            data = inverted_index.get(var, [0])  
            index_dict[var] = set(data[1])  
        else:  
            index_dict[var] = set()  
    return index_dict
```

`evaluate(expr, assignment)`

评估给定布尔表达式的值。

- **参数**:
 - `expr`: 布尔表达式的字符串。
 - `assignment`: 映射变量为布尔值的字典。
- **返回**: 布尔表达式的评估结果 (`True` 或 `False`)。

```
def evaluate(expr, assignment):
    """评估布尔表达式的值"""
    expr = expr.replace(" AND ", " and ").replace(" OR ", " or ").replace(" NOT ", " not ")
    for var, val in assignment.items():
        expr = expr.replace(var, str(val))
    return eval(expr)
```

`bool_query_optimized(expr, vars, index_dict)`

优化的布尔查询实现。

- **参数:**
 - `expr`: 优化后的布尔表达式字符串。
 - `vars`: 表达式中使用的变量列表。
 - `index_dict`: 每个变量及其对应文档索引的集合。
- **返回:** 满足条件的文档索引的列表。

```
def bool_query_optimized(expr, vars, index_dict):
    # 求index_dict中所有索引集合的并集
    result_set = set()

    # 生成所有可能的真值组合
    for assignment in product([False, True], repeat=len(vars)):
        assignment_dict = dict(zip(vars, assignment))
        result = evaluate(expr, assignment_dict)
        temp_set = set.union(*index_dict.values())

        # 如果结果为真，则收集该组合的子句
        if result:
            # 判断是否每个变量都为假
            if all(not assignment_dict[v] for v in vars):
                # 报错
                raise ValueError("表达式不合法")
            for var in vars:
                if assignment_dict[var]: # 如果变量为真，加入原变量
                    # 求temp_set和index_dict交集
                    temp_set = temp_set.intersection(index_dict[var])
            for var in vars:
                if not assignment_dict[var]: # 如果变量为假，加入NOT原变量
                    # 求temp_set和index_dict相减
                    temp_set = temp_set.difference(index_dict[var])

            # 求temp_set和result_set并集
            result_set = result_set.union(temp_set)

    return list(result_set)
```

`execute_queries(expression, synonym_dict, inverted_index)`

执行优化的布尔查询并返回结果。

- **参数:**
 - `expression`: 输入的布尔查询表达式。

- `synonym_dict`: 用于变量同义词替换的字典。
- `inverted_index`: 存储反向索引数据的字典。
- **返回**: 优化后的布尔查询结果的列表。

```
def execute_queries(expression, synonym_dict, inverted_index):
    vars, new_expression = extract_variables(expression, synonym_dict) # 在表达式
    # 中出现的变量
    index_dict = generate_index_dict(inverted_index, vars) # 生成索引字典
    dnf_result = bool_query_optimized(new_expression, vars, index_dict)
    return dnf_result
```

list_storage.py

实验任务中索引压缩的部分在 `lab1-1/list_storage.py` 中实现，以下是关键代码说明：

`front_encoding(ids)`

使用间距代替文档 ID。

- **参数**:
 - `ids`: 原始文档 ID 列表。
- **返回**: 压缩后的文档 ID 列表。
- **描述**: 如果 ID 列表为空，返回一个空列表。否则，将第一个 ID 加入结果列表，其余 ID 通过减去前一个 ID 进行压缩。

```
def front_encoding(ids):
    if not ids:
        return []
    # 使用差分编码
    encoded = [ids[0]] # 加入第一个ID
    for i in range(1, len(ids)):
        encoded.append(ids[i] - ids[i - 1])
    return encoded
```

`front_encoding_index(inverted_index)`

对倒排索引进行差分压缩。

- **参数**:
 - `inverted_index`: 原始倒排索引，字典类型。
- **返回**: 压缩后的倒排索引。
- **描述**: 使用差分编码对每个标签的文档 ID 列表进行编码，并返回压缩后的倒排索引。

```
def front_encoding_index(inverted_index):
    compressed_index = {}
    for tag, (frequency, ids) in inverted_index.items():
        # 计算间距
        gaps = front_encoding(ids)
        compressed_index[tag] = [frequency, gaps]
    return compressed_index
```

`compress_inverted_index(inverted_index)`

对倒排索引进行可变长度编码压缩。

- **参数:**
 - `inverted_index`: 原始倒排索引, 字典类型。
- **返回:** 压缩后的倒排索引。
- **描述:** 使用差分编码和可变长度编码对倒排索引中的每个文档 ID 列表进行压缩, 并返回结果。

```
def compress_inverted_index(inverted_index):
    compressed_index = {}
    for tag, (frequency, ids) in inverted_index.items():
        # 计算间距
        gaps = front_encoding(ids)
        # 使用可变长度编码进行压缩
        encoded_gaps = variable_length_encode(gaps)
        compressed_index[tag] = [frequency, encoded_gaps]
    return compressed_index
```

`variable_length_encode(gaps)`

对 ID 进行可变长度编码。

- **参数:**
 - `gaps`: 要编码的间距列表。
- **返回:** 编码后的字节流。
- **描述:** 根据间距值进行编码, 低于 128 的值采用 1 位延续位, 高于 128 的值采用多个字节表示。

```
def variable_length_encode(gaps):
    byte_array = bytearray()
    for g in gaps:
        if g < 128:
            # G < 128
            byte_array.append(g | 0x80) # 1位延续位为1
        else:
            # G >= 128
            while g >= 128:
                byte_array.append(g & 0x7F) # 低7位
                g >= 7 # 右移7位
            # print(g | 0x80)
            byte_array.append(g | 0x80) # 最后的字节
    return bytes(byte_array)
```

`variable_length_decode(encoded_bytes)`

对编码后的字节流进行解码。

- **参数:**
 - `encoded_bytes`: 编码后的字节流。
- **返回:** 解码后的间距列表。
- **描述:** 将编码的字节流还原为原来的间距列表。

```
def variable_length_decode(encoded_bytes):
    gaps = []
```

```

i = 0
current_gap = 0
k = 0
while i < len(encoded_bytes):
    byte = encoded_bytes[i]
    current_gap += (byte & 0x7F) << (7 * k) # 低7位
    if(byte & 0x80 == 0):
        i += 1
        k += 1
        continue
    else:
        gaps.append(current_gap)
        current_gap = 0
        k = 0
    i += 1
return gaps

```

`decompress_inverted_index(compressed_index)`

解压缩倒排索引。

- **参数:**
 - `compressed_index`: 压缩后的倒排索引。
- **返回:** 原始倒排索引。
- **描述:** 还原压缩后的倒排索引，构建出原始的文档 ID 列表。

```

def decompress_inverted_index(compressed_index):
    inverted_index = {}
    for tag, (frequency, encoded_gaps) in compressed_index.items():
        gaps = variable_length_decode(encoded_gaps)
        ids = [gaps[0]] # 第一个间距直接作为id
        # 根据间距重建ID列表
        for gap in gaps[1:]:
            ids.append(ids[-1] + gap)

        inverted_index[tag] = [frequency, ids]
    return inverted_index

```

`save_index(compressed_index, filename)`

存储倒排索引到压缩文件。

- **参数:**
 - `compressed_index`: 压缩后的倒排索引。
 - `filename`: 输出文件名。
- **返回:** 无。
- **描述:** 将倒排索引以二进制形式保存到指定文件中。

```

def save_index(compressed_index, filename):
    with open(filename, 'wb') as file:
        pickle.dump(compressed_index, file)

```

```
load_index(filename)
```

从压缩文件中读取倒排索引。

- **参数:**
 - `filename`: 文件名。
- **返回:** 读取的倒排索引。
- **描述:** 从指定文件中加载压缩后的倒排索引，并返回给调用者。

```
def load_index(filename):  
    with open(filename, 'rb') as file:  
        return pickle.load(file)
```

结果分析

预处理效果分析

分词结果对比

在实验中使用三种方式进行分词，分别为双向最大匹配分词，结巴分词和 pkuseg，将分词结果分别导出到以下位置：

1. 使用 jieba 从电影数据中提取标签并导出到: `lab1-1/dataset/movie_tag.csv`
2. 使用 jieba 从书籍数据中提取标签并导出到: `lab1-1/dataset/book_tag.csv`
3. 使用 pkuseg 对电影数据提取标签并导出到: `lab1-1/dataset/movie_tag_pkuseg.csv`
4. 使用 pkuseg 对书籍数据提取标签并导出到: `lab1-1/dataset/book_tag_pkuseg.csv`
5. 使用双向最大匹配算法对电影数据导出标签到: `lab1-1/dataset/movie_tag_maxmatch.csv`
6. 使用双向最大匹配算法对书籍数据导出标签到: `lab1-1/dataset/book_tag_maxmatch.csv`

在 `compare.py` 中，将分词结果进行词频统计，计算词频向量，将词频向量转换为向量矩阵，两两之间计算余弦相似度，得到结果如下：

```
PS E:\third\Web\webinfo> & C:/Users/86189/.conda/envs/Webinfo/python.exe e:/third/Web/webinfo/lab1-1/compare.py  
jieba-pkuseg: 0.97046826  
jieba-maxmatch: 0.96847749  
pkuseg-maxmatch: 0.93212711
```

由相似度结果可知，不同分词结果之间的差异较小；具体来说，采用现有的分词工具（jieba 和 pkuseg）得到的分词结果之间差异最小，而使用双向最大匹配算法得到的分词结果与前两者差异稍大，但总体差异依然较小。这说明在选用合适的词典（本实验双向最大匹配算法使用的词典来自 jieba）时，使用双向最大匹配算法得到的分词结果的准确性已经很高了。

同义词替换效果分析

在进行信息检索时，用户可能会使用不同的词汇来表达相同的意思。如果没有同义词替换，检索时可能无法命中包含同义词的文档，造成相关信息无法被找到，从而降低了检索的准确性和全面性。例如，不使用同义词替换时的倒排表的部分如下：

```
83757      "史观": [  
83758          2,  
83759          [  
83760              1015699,  
83761              1427825  
83762          ]  
83763      ],  
83764      "史记": [  
83765          2,  
83766          [  
83767              1077847,  
83768              1844629  
83769          ]  
83770      ],  
83771      "史论": [  
83772          7,  
83773          [  
83774              1003284,  
83775              1017818,  
83776              1034067,  
83777              1064271,  
83778              1941558,  
83779              2294993,  
83780              3333989  
83781          ]  
83782      ],  
83783      "史评": [  
83784          5,  
83785          [  
83786              1003479,  
83787              1025723,  
83788              1041482,  
83789              1050175,  
83790              1858410  
83791          ]  
83792      ],
```

可以看到多个意思相近的词语并没有被合并，而是分别建立了各自的索引表，这将导致布尔查询时查询结果的局限性。例如，在面对布尔查询（（人工智能 AND NOT（机器人 OR 自动化））OR（大数据 AND（分析 OR 可视化）））AND（技术 OR 创新）时，不使用同义词替换时查询结果为空，而使用同义词替换后，查询结果为 [1003479, 1008145, 1021273, 1056461, 1083762, 1203426, 1396339, 1440885, 1867642, 1868786, 1926103, 1949338, 2052448, 2253642, 3000997, 3009821, 3274113, 3344676, 3626924, 3813669, 4618225, 4872671]，二者差别明显。

删除停用词效果分析

停用词通常是指一些在语义上不携带重要信息的词汇，如“的”、“是”、“在”、“有”等。如果不删除这些停用词，倒排索引表将会变得非常庞大，存储和处理这些冗余信息会浪费存储空间和增加计算复杂度。例如，不删除停用词时的倒排表的部分如下：

```
1  {  
2  "的": [  
3      1192,  
4  >  [...  
1197  ]  
1198  ]  
1199  }
```

可以看到词语“的”的倒排索引表长度为 1192 行，这与总的书籍数目几乎相同。诸如此类的无意义词语，带来了大量的信息冗余，浪费了存储空间。

布尔查询效率分析

使用多种测试样例，对两种不同的布尔查询的实现，每个样例分别进行 10 次查询，计算总耗时，得到如下输出结果：

对于不进行优化的的布尔查询，查询效率如下：

查询语句: (动作 AND 剧情) OR (科幻 AND NOT 恐怖)
查询耗时: 0.126482秒
查询语句: ((山脉 OR 海洋) AND (夏天 AND NOT 雨天)) OR (城市 AND (夜景 OR 商业))
查询耗时: 0.126267秒
查询语句: ((学习 AND (编程 OR 数据分析)) AND (工具 OR 资源))
查询耗时: 0.000141秒
查询语句: ((疫情 OR 疫苗) AND NOT (恐慌 OR 死亡)) OR (健康 AND (生活方式 OR 饮食))
查询耗时: 0.119688秒
查询语句: ((人工智能 AND NOT (机器人 OR 自动化)) OR (大数据 AND (分析 OR 可视化))) AND (技术 OR 创新)
查询耗时: 0.123268秒
查询语句: (小说 AND (爱情 OR 冒险)) OR (非虚构 AND (历史 OR 传记))
查询耗时: 0.001337秒
查询语句: ((海明威 OR 莎士比亚) AND (小说 OR 诗歌)) AND (经典 OR 现代)
查询耗时: 0.000949秒
查询语句: (儿童 AND 图画书 AND (教育 OR 娱乐)) OR (青少年 AND 小说 AND (成长 OR 探险))
查询耗时: 0.001082秒
查询语句: ((英语 AND (原版 OR 翻译)) AND NOT (外语学习 OR 教材)) OR (文学 AND 科幻)
查询耗时: 0.117687秒
查询语句: ((悬疑 AND (推理 OR 侦探)) AND NOT (恐怖 OR 血腥)) OR (奇幻 AND (魔法 OR 冒险))
查询耗时: 0.119044秒

对于使用真值表进行优化的布尔查询，查询效率如下：

查询语句: (动作 AND 剧情) OR (科幻 AND NOT 恐怖)
查询耗时: 0.002802秒
查询语句: ((山脉 OR 海洋) AND (夏天 AND NOT 雨天)) OR (城市 AND (夜景 OR 商业))
查询耗时: 0.025281秒
查询语句: ((学习 AND (编程 OR 数据分析)) AND (工具 OR 资源))
查询耗时: 0.005310秒
查询语句: ((疫情 OR 疫苗) AND NOT (恐慌 OR 死亡)) OR (健康 AND (生活方式 OR 饮食))
查询耗时: 0.021670秒
查询语句: ((人工智能 AND NOT (机器人 OR 自动化)) OR (大数据 AND (分析 OR 可视化))) AND (技术 OR 创新)
查询耗时: 0.058761秒
查询语句: (小说 AND (爱情 OR 冒险)) OR (非虚构 AND (历史 OR 传记))
查询耗时: 0.059171秒
查询语句: ((海明威 OR 莎士比亚) AND (小说 OR 诗歌)) AND (经典 OR 现代)
查询耗时: 0.047823秒
查询语句: (儿童 AND 图画书 AND (教育 OR 娱乐)) OR (青少年 AND 小说 AND (成长 OR 探险))
查询耗时: 0.225065秒
查询语句: ((英语 AND (原版 OR 翻译)) AND NOT (外语学习 OR 教材)) OR (文学 AND 科幻)
查询耗时: 0.102182秒
查询语句: ((悬疑 AND (推理 OR 侦探)) AND NOT (恐怖 OR 血腥)) OR (奇幻 AND (魔法 OR 冒险))
查询耗时: 0.080065秒

二者进行对比可以发现：

对于不进行优化的布尔查询实现，如果查询条件中没有 NOT，此时不需要遍历整个索引表 S，只需要查找布尔变量对应的索引表查询速度很快；如果查询条件中出现 NOT，此时需要遍历整个索引表 S 来求布尔变量的补集，导致查询速度大大降低。而由于使用栈来从左向右分析查询条件，布尔变量的数量多少并不会显著影响到查询速度；




而对于使用真值表进行优化的布尔查询实现，由于遍历了真值表中的所有表项，所以随着布尔变量数量的增加，查询耗时呈指数级增加，当布尔变量达到 8 个以上时，会全面慢于不进行优化的布尔查询实现；但当布尔变量较少时，由于只需要布尔变量对应的索引字典，而不需要遍历整个索引表 S，在面对查询条件中出现 NOT 的情况时，查询速度显著快于不进行优化的布尔查询实现。同时，由于只需要布尔变量对应的索引字典，优化后的布尔查询实现占用的内存空间显著降低。

索引压缩效果对比

主要分为存储空间的对比和检索效率的对比

存储空间对比

以 Book 类数据为例，如下图所示：

 compressed_index.pkl	2024/11/18 15:58	PKL 文件	620 KB
 front_encoded_index.pkl	2024/11/18 15:58	PKL 文件	783 KB
 index.pkl	2024/11/18 15:58	PKL 文件	1,015 KB

对于不进行任何压缩的倒排索引表，打包后占用空间大小为 1015kB；

在使用间距代替文档 ID 后，打包后占用空间大小为 783kB，压缩了 22.9%；

进一步，对间距进行可变长度编码后，打包后占用空间大小为 620kB，进一步压缩了 20.8%，对比不进行任何压缩的倒排索引表，占用空间仅为原来的 61.1%，达到了较好的压缩效果。

检索效率对比

由于实验的实现中，两种复杂布尔查询的实现都需要首先将倒排索引表 S 加载到内存中，故是否压缩不会直接影响最后的检索效率；但在布尔查询初始化时，需要对压缩后的倒排索引表进行解压，而压缩程度越大的压缩方式，其解压所需时间也越久，从而间接导致检索速度的下降。

第 2 阶段 豆瓣数据的个性化检索与推荐

实验任务

基于豆瓣 Movie&Book 的 tag 信息、我们提供的豆瓣电影与书籍的评分记录以及用户间的社交关系，判断用户的偏好。

实验内容

- 采用协同过滤方式，仅利用用户-项目（电影或书籍）的评分矩阵进行评分预测。
- 根据提供的 tag 等文本信息进行辅助预测，辅助形式自行选择（如：使用 tag 补充书籍的信息）。
- 基于社交网络关系的推荐。

实验方法

基本思想

基础实验

假定用户偏好不与时间有关，即如果用户给某一类的书籍/电影打高分，我们就认为他会给此类相似的书籍/电影打高分。

此处我们给出两种计算预测评分的方法：基于 KNN 的评分预测方法和线性拟合方法。

基于项目推荐的协同过滤

对于用户评价过的某一本书籍/电影，我们可以找到与其最为相似的 k 本书籍/电影，计算该用户对这 k 本书籍/电影的评分的加权平均值作为预测评分。

获取该用户对其他项目的评分信息，计算需要预测项目与其他项目的相似度，通过基于相似度的 KNN 方法预测该项目的评分，对候选项目依据评分进行排序以展示用户偏好

$$\text{rating} = \frac{\sum_{i=1}^k \text{rating}_i \cdot \text{similarity}_i}{\sum_{i=1}^k \text{similarity}_i}$$

基于相似度的线性回归

同样使用相似度，将相似度与对应评分进行线性回归，损失函数设置为与正确评分的平均平方差，得到预测模型

此时我们划分训练集和测试集，划分比例为 1:9，使用训练集训练线性回归模型，然后使用测试集计算 MSE 和 NDCG。

$$\text{rating} = \sum_{i=1}^n w_i \cdot \text{feature}_i$$

选做实验

基本思想相似，爬取用户社交网络中其他人对项目的评分，计算相似度，得到基于社交网络的评分信息，将基于社交网络的评分预测与用户自身的评分预测进行加权平均，探索极值，获取性能最优越的评分预测方法

关键代码说明

项目相似程度的度量

通过实验一第一阶段获得的 booktags 数据（即对项目的所有评价进行分词，近义词规划）得到该项目的 tags 表，计算所有项目和在所有词项下的 tf-idf 评分，将评分数据作为项目的特征向量，计算不同项目的特征向量的余弦相似度来获得任意两个向量的相似度

计算 tfidf

```
data['Tags'] = data['Tags'].apply(lambda x: ' '.join(eval(x)))

# 初始化 TfidfVectorizer
vectorizer = TfidfVectorizer()

# 对 'Tags' 列进行 TF-IDF 分析
tfidf_matrix = vectorizer.fit_transform(data['Tags'])

# 获取词汇表
feature_names = vectorizer.get_feature_names_out()

# 将 TF-IDF 矩阵转换为 DataFrame，并添加书的 ID 列
tfidf_df = pd.DataFrame(tfidf_matrix.toarray(), columns=feature_names)
tfidf_df.insert(0, 'Book', data['Book'])
```

获得了每个项目的特征向量表示

相似度计算

```
def cal_similarity(books, rates, book, tfidf_df):

    if books == []:
        return
```

```

book_vector1 = tfidf_df[tfidf_df['Book'] == idx_to_book.get(book)].iloc[:,
1:].values

sim_scores = []

for j in range(len(books)):
    if rates[j] == 0:
        continue
    book_vector2 = tfidf_df[tfidf_df['Book'] ==
idx_to_book.get(books[j])].iloc[:, 1:].values
    similarity = cosine_similarity(book_vector1,book_vector2)
    sim_scores.append([str(idx_to_book.get(books[j])), similarity[0][0],
rates[j]])
    return sim_scores

```

获取两本书得特征向量表示，计算余弦相似度

历史数据获取

获取相同用户过往的所有评分数据

计算相同用户所有评分项目两两之间的相似度

```

u_items_list[str(idx_to_user.get(user))] = []

for i in range(len(books)):
    if rates[i] == 0:
        continue

u_items_list[str(idx_to_user.get(user))].append([str(idx_to_book.get(books[i]))
, int(rates[i]), []])
    for j in range(len(books)):
        if rates[j] == 0:
            continue
        similarity = similarity_matrix[i][j]
        u_items_list[str(idx_to_user.get(user))][-1]
[2].append((int(rates[j]), float(similarity)))

```

由此获得一个项目与用户历史交互数据的关系和表示

社交网络遍历

获取社交网络

```

user_score = score[score['User'] == user_id]

# 读取用户的朋友信息

contacts = [contact.strip().split(':') for contact in contacts]
contacts = {contact[0]: contact[1].split(',') for contact in contacts}

# 计算用户的朋友的评分信息
friends_score = []
for friend in contacts[user_id]:
    friend_score = score[score['User'] == int(friend)]
    if friend_score.empty:
        continue

```

```
friends_score.append(friend_score)
```

计算用户的朋友的评分信息的倒排索引

```
for friend_score in friends_score:
    for index, row in friend_score.iterrows():
        book_id = row['Book']
        rate = row['Rate']
        if book_id in user_score['Book'].values: # 如果用户已经评分过该书籍，则不
考虑
            continue
        if book_id in friends_inverted_index:
            friends_inverted_index[book_id].append(rate)
        else:
            friends_inverted_index[book_id] = [rate]
```

获取评分项目的特征向量

```
fri_rating, fri_books = [], []
for i, rates in friends_inverted_index.items():
    for r in rates:
        if r == 0:
            continue
        fri_rating.append(r)
        fri_books.append(i)
filtered_df = tfidf_df[tfidf_df['Book'].isin(fri_books)]
ordered_vectors = filtered_df.set_index('Book').loc[fri_books].iloc[:,
1:].values
```

与上一部分相同的相似度计算

由此获得一个项目与社交网络交互数据的关系和表示

K 近邻分析

分别计算两部分交互数据表示的 k 近邻

具体方法是选取交互数据表示中与本项目相似度最高的前 k 个

```
def slicing(ratings, similarities, k):
    sliced_ratings = []
    sliced_similarities = []
    for rating, similarity in zip(ratings, similarities):
        if len(rating) < k:
            sliced_ratings.append(rating)
            sliced_similarities.append(similarity)
        else:
            combined = list(zip(rating, similarity))
            combined.sort(reverse=True, key=lambda x: x[1])
            rating, similarity = zip(*combined[:k])
            sliced_ratings.append(rating)
            sliced_similarities.append(similarity)
    return sliced_ratings, sliced_similarities
```

利用相似度对这 k 个评分数据加权平均

```
answer_ratings = [np.dot(r,s)/sum(s) for r,s in zip(ratings,similarities)]
```

分别获取**基于历史评分数据**的评分预测和**基于社交网络评分数据**的评分预测，其中前者即为该阶段的基础实验内容

加权平均

将两种评分数据按一定参数加权平均，根据实验结果，探索加权平均收益的极大值

```
if FRIEND and fri_ratings != []:
    fri_ratings, fri_similarities =
slicing(fri_ratings, fri_similarities, k)
    answer_fri_ratings = [np.dot(r,s)/sum(s) for r,s in
zip(fri_ratings, fri_similarities)]
    answer_ratings = lam * np.array(answer_ratings) + (1-lam) *
np.array(answer_fri_ratings)
```

线性回归

将特征值定义为历史交互记录的评分和相似度，padding 补足到定长

```
max_len = max(len(r) for r in ratings)
ratings_padded = np.array([r + [int(0)] * (max_len - len(r)) for r in ratings])
similarities_padded = np.array([s + [int(0)] * (max_len - len(s)) for s in
similarities])

features = np.hstack(similarities_padded)

features = [np.dot(r, s) / sum(s) for r, s in zip(ratings_padded,
similarities_padded)]
features = np.array(features).reshape(-1, 1)
target_ratings = np.array(target_ratings)
```

划分训练和测试集合，进行线性回归分析

```
X_train, X_test, y_train, y_test = train_test_split(features, target_ratings,
test_size=0.2, random_state=42)

# 训练线性回归模型
model = LinearRegression()
model.fit(X_train, y_train)
```

在测试集上运行训练成果

```
ratings_padded = np.array([r + [int(0)] * (max_len - len(r)) for r in
ratings])
similarities_padded = np.array([s + [int(0)] * (max_len - len(s)) for s in
similarities])
new_features = [np.dot(r, s) / sum(s) for r, s in zip(ratings_padded,
similarities_padded)]
new_features = np.array(new_features).reshape(-1, 1)
answer_ratings = model.predict(np.array(new_features))
```


性能测试方法

采用 MSG 和 NDCG 两种度量来评估实验结果

MSG

```
mean_squared_error(target_sort, answer_sort)
```

NDCG

统一对结果的前五位做利用用户真实评分信息做 NDCG 分析，获取排序效果

```
def dcg(scores):  
  
    return np.sum([(2**score - 1) / np.log2(idx + 2) for idx, score in  
enumerate(scores)])  
  
def ndcg(target_sort, answer_sort):  
    ndcg = []  
    for i in range(len(target_sort)):  
        dcg_val = dcg(answer_sort[:i+1])  
        idcg_val = dcg(target_sort[:i+1])  
        ndcg.append(dcg_val / idcg_val if idcg_val > 0 else 0)  
    return ndcg
```

结果分析

kNN 与 Linear Regression 的结果对比

此处的均采用不包含社交网络的评分预测结果

	MSE	NDCG
kNN	0.6537304444906865	0.7991238148683567
Linear Regression	0.6429722291005368	0.8012059320420349

在这里我们可以看到，线性回归的效果略好于 kNN 方法。NDGC 的值在 0.8 左右，说明我们的预测效果还是不错的。

基于历史评分数据的评分预测 和 基于社交网络评分数据的评分预测

最终的评分数据由基于历史评分数据的评分预测 和 基于社交网络评分数据的评分预测 加权平均而来

其中有两个关键参数：

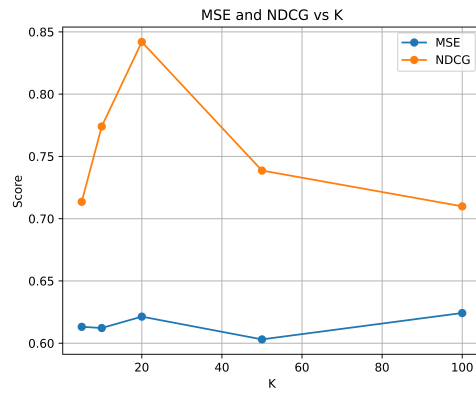
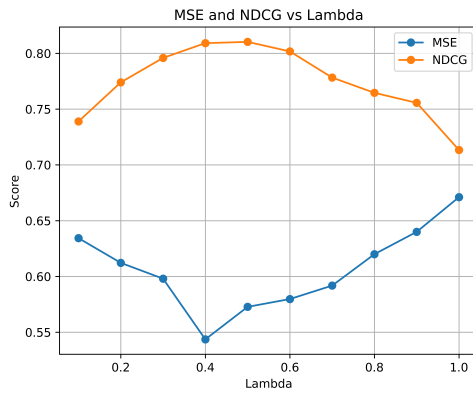
- 两种预测数据加权平均参数

λ

- kNN 的控制参数，即选取 TOP 相关度数据的个数

k

分别控制 λ 在 0~1, k 在[5,10,20,30,40,50]浮动 获取评分预测的表现



可以看出评分在 $\text{lambd} = 0.4$, $k = 20$ 时取最高值

由此得到我们基于协同过滤的评分推荐系统最终方案

利用最终方案运行，得到实验表现：

```
(watermark) PS D:\work\InternetTechnology\24webinfo\webinfo> & 'd:\Anaconda3\envs\watermark\python.exe' '-c':Users\林帆.vscode\extensions\ms-python.debugpy-2024.12.0-win32-x64\bundled\libs\debugpy\adapter\...\debugpy\launcher' '5624' '--' 'd:\work\InternetTechnology\24webinfo\webinfo\lab1-2knn.py'
```

```
NDCG: 0.64893949712124066, MSE : 0.880857529617123
```

| 20/20 [05:27<00:00, 16.38s/it]

NDCG: 0.6489394972124066, MSE : 0.8800557529617123