Web LAb1

组员：

陆文博　PB22000135

岳梓烨　PB22000159

陈禾一　PB22000176

Part 1

Section 1 分词

我是用了两种现成的分词工具，并自己加入了一些分词优化以使得分词的结果更符合后续实验要求。两种的现成的分词工具分别为jieba分词和pkuseg分词。完成代码时我并没有特别考虑到拓展性，所以我使用的时面向过程编程，分别对book和movie使用jieba分词和pkuseeg分词，这里我们使用jieba分词对book的tag的分词为例子。

```
T1 = time.time()
for i,member in enumerate(book_reader):
    if(i==0): continue
    is_pair = 0
    str = ""

    book_output_row = ""
    book_output_row += member[0] + "," + '"' + "{"

    words = []
    # print(member[1])
    for ch in member[1]:
        if((ch == "," or ch == '}')and is_pair == 1):
            is_pair = 0
            words.append(str[:-1])
            str = ""
            continue
        if(is_pair == 1):
            str += ch
        if(ch == "'" and is_pair == 0): is_pair = 1
    token = []
    for word in words:
        tokens = ','.join(jieba.cut(word, HMM=True))
        for token in tokens.split(','):
            f = 0
            for bad in bad_list:
                if(token == bad):
                    f = 1
                    break
            if(f == 0):
                book_output_row += "'" + token + "'" + ","
    book_output_row += "}" + '"' + "\n"
    book_content.append(book_output_row)

T2 = time.time()
print("jieba book time: ", T2-T1)

with open(book_output_path, 'w') as f:
    f.write(book_column + "\n")
    f.writelines(book_content)
```

具体的分词方式我们使用调包完成，代码的中间部分是对源csv文件的读取。原来的Tags使用的是一个由单引号、逗号、Tag等组成的字符串，这里把其中真正的Tag进行抽取并进行分词，最终以相同的格式重新储存到文件里面。其中值得一提的是，我们根据前期工作分词的结果，筛选出了我们不想要的一些代表不了信息的词汇汇总为bad_list，在提取时碰到这些词汇不会加入到分词表中。

接下来分析Jieba分词和PKUSeg分词

Jieba 分词基于 词典+HMM（隐马尔科夫模型） 的统计方法，使用双向最大匹配（Bi-MM）来匹配词典中的词语。HMM模型用于处理未登录词（未出现在词典中的词），通过字词的状态转移概率来预测分词边界。(本次实验中我们并未使用jieba分词的paddle模式，即使用paddle深度学习框架，我们也没能使用jieba分词的自定义字典)

PKUSeg 分词基于条件随机场（CRF）或深度学习模型的序列标注方法。使用大规模标注语料训练词边

界的概率分布，通过全局最优来预测分词结果。

```
(web) sakiko@LAPTOP-9TSVN48O:~/Web/lab/web-lab1/src_lwb$ python cutter_jieba.py
Building prefix dict from the default dictionary ...
Loading model from cache /tmp/jieba.cache
Loading model cost 0.494 seconds.
Prefix dict has been built successfully.
jieba book time:  4.415745973587036
jieba movie time:  16.06471824645996
(web) sakiko@LAPTOP-9TSVN48O:~/Web/lab/web-lab1/src_lwb$ python cutter_pkuseg.py
pkuseg book time:  10.418039798736572
pkuseg movie time:  46.36260175704956
```

1. 性能对比
   准确性: PKUSeg 在复杂场景和领域适配上表现优异，因为其模型能捕捉上下文和细粒度语义。
   速度: Jieba 因为算法简单，分词速度更快，适合大规模实时处理任务。
   未登录词处理: Jieba 依赖 HMM，可能效果有限；而 PKUSeg 的深度学习方法能够更好地识别新词。
2. 性能差异原因
   算法复杂度: Jieba 是启发式规则与简单统计方法，性能上更轻量；PKUSeg 基于训练数据和深度学习模型，对算力要求更高。
   数据依赖: PKUSeg 需要高质量标注数据进行训练，适应性强；而 Jieba 的性能主要受词典的质量限制。
   上下文建模能力: PKUSeg 的深度学习模型能有效捕捉上下文关系；Jieba 的 HMM 模型对于长距离依赖能力较弱。

Section 2

通过上一阶段生成的分词结果，生成倒排索引表，处理过程中采用{(word，[id1, id2...])...}的格式存储。

首先通过基础方式存储，即对于词典，采取[88byte词项，4byte频率，4byte倒排索引指针]方式顺序存储（发现上述分词结果中最长的词项为85bytes，所以采用88bytes存储，空位使用b'\x00'填充）。

对于倒排索引文件，采用4字节每文档id的方法顺序存储。

（上述生成倒排索引表和词典文件的方法在"web-lab1\src_yzy\inverted_index_gen.py"中，采用基本顺序存储的读取方法测试（从二进制文件还原倒排索引表）包含在"web-lab1\src_yzy\test_basic.py"中）

具体建立倒排索引表的过程如下(in web-lab1\src_yzy\inverted_index_gen.py)：

```python
def insert_line(i, textline):
    global inverted_index
    if i == 0:
        return

    index = int(textline[0])    #book id
    words = []                  #tags
    is_pair = 0
    str = ""

    for ch in textline[1]:
        if((ch == "," or ch == '}')and is_pair == 1):
            is_pair = 0
            words.append(str[:-1])
            str = ""
            continue
        if(is_pair == 1):
            str += ch
        if(ch == "'" and is_pair == 0): is_pair = 1
    for word in words:
        set = inverted_index.setdefault(word, [])   #不使用三步走，直接使用字典+列表
```

```
        set.append(index)
```

用于处理一行对应的记录。

基础的顺序存储实现如下(in web-lab1\src_yzy\inverted_index_gen.py)(去除了编写时的一些测试项):

```
def write_index_normal(file_path):  #需要分开存储词典和倒排表项
    global inverted_index
    #测试得到最长词项为85字节，于是取88字节存储词项
    dict_pt = 0
    index_pt = 0
    file1 = open(file_path + "_basic_store_dict.binery", "wb")
    file2 = open(file_path + "_basic_store_index.binery", "wb")
    for index,(key, value) in enumerate(inverted_index):
        word = key.encode() + (88 - len(key.encode())) * b'\x00'
        file1.write(word)
        dict_pt += 88
        file1.seek(dict_pt)
        file1.write(len(value).to_bytes(4))
        dict_pt += 4
        file1.seek(dict_pt)
        file1.write(index_pt.to_bytes(4))
        dict_pt += 4
        file1.seek(dict_pt)
        for id in value:
            file2.write(id.to_bytes(4))
            index_pt += 4
            file2.seek(index_pt)
    return
```

与之对应的，从二进制文件中还原的过程如下(in web-lab1\src_yzy\test_basic.py):

```
input = ["../data/index/selected_book_top_1200_data_tag_tokenized_jieba",
"../data/index/selected_book_top_1200_data_tag_tokenized_pkuseg",
"../data/index/selected_movie_top_1200_data_tag_tokenized_jieba",
"../data/index/selected_movie_top_1200_data_tag_tokenized_pkuseg"]
output =
["../data/index/testread/selected_book_top_1200_data_tag_tokenized_jieba",
"../data/index/testread/selected_book_top_1200_data_tag_tokenized_pkuseg",
"../data/index/testread/selected_movie_top_1200_data_tag_tokenized_jieba",
"../data/index/testread/selected_movie_top_1200_data_tag_tokenized_pkuseg"]

def process(i):
    file_dict = open(input[i] + "_basic_store_dict.binery", "rb")
    file_index = open(input[i] + "_basic_store_index.binery", "rb")
    file_dict_out = open(output[i] + "_dict.txt", "w", encoding="utf-8")
    file_index_out = open(output[i] + "_index.txt", "w", encoding="utf-8")

    dict_data = file_dict.read()
    index_data = file_index.read()

    dict_ptr = 0
    index_ptr = 0
    print(len(dict_data))
    while (dict_ptr < len(dict_data)):
        word = (dict_data[dict_ptr : dict_ptr +
88].rstrip(b'\x00')).decode('utf-8')
```

```
        freq = int.from_bytes(dict_data[dict_ptr + 88 : dict_ptr + 92])
        pindex = int.from_bytes(dict_data[dict_ptr + 92 : dict_ptr + 96])
        dict_ptr += 96
        print(word, freq, pindex, file=file_dict_out)
        index = []
        for i in range(0, freq):
            index.append(int.from_bytes(index_data[index_ptr : index_ptr + 4]))
            index_ptr += 4
        print(index, file=file_index_out)
```

实际上除了个别长词之外，一般的词项远远达不到88字节，我们考虑对词典文件进行压缩。除去基本存储方法以以外，试图采用单一字符串词典以及按块存储方法，并比较这三种方法的空间占用以及检索效率。

单一字符串方法采用：开头存储二进制长字符串字节数，然后存放对应大小的字符串，接下来每12字节中依次存放{4字节字符串指针，4字节词项频率，4字节倒排索引表指针}

按块存储采用：开头存放字符串方法相同，然后每5项存放一次字符串指针，接着对于每一项的9字节：{1字节词项字符串长度，4字节频率，4字节倒排索引表指针}

相关的存放代码以及读取方式在如下文件中实现：

"web-lab1\src_yzy\word_compress_store_single.py"

"web-lab1\src_yzy\word_compress_store_block.py"

"web-lab1\src_yzy\test_single.py"

"web-lab1\src_yzy\test_block.py"

关于空间占用，结果如下：

（首先在第一阶段中，我们对于book和movie使用两种工具进行了分词，以下是共四组倒排索引表使用三种词典压缩方法后的空间占用对比，相应的倒排索引表文件大小也包括在内，由于对倒排索引表没有压缩因此大小不存在变动）

	book_jieba	book_pkuseg	movie_jieba	movie_pkuseg
顺序存储	1836KB	1844KB	4958KB	5780KB
单一字符串	355KB	363KB	984KB	1230KB
按块存储	313KB	320KB	870KB	1098KB
倒排索引表	1180KB	1184KB	4817KB	4512KB

从上面的结果可以看出，从简单的顺序存储（每一项使用88字节保证存储超长词汇）到单一字符串的改进大大缩小了词典文件的存储，可见大量的短字符串是达不到88字节的长度的。

从单一字符串到按块索引的改进进一步缩小了词典文件的大小（在这里设置每5个块放置一次字符串指针），理论上除了开始的长字符串可以每60字节节省12字节，实际的表现也非常符合预期结果。

Section 3 布尔检索

接下来我们需要实现对倒排索引表的布尔查询。

布尔检索是数据库检索最基本的方法，主要使用逻辑“或”（+、OR）、逻辑"与"(×、AND)、逻辑"非"(- 、NOT)三种运算。

在本次实验中，我们可以通过实现And， Or， And Not 三个函数来实现倒排表的布尔查询。

```python
def Or(list1, list2):
    i, j = 0, 0
    res = []
    while i < len(list1) and j < len(list2):
        # 同时出现，只需要加入一次
        if list1[i] == list2[j]:
            res.append(list1[i])
            i += 1
            j += 1
        # 指向较小数的指针后移，并加入列表
        elif list1[i] < list2[j]:
            res.append(list1[i])
            i += 1
        else:
            res.append(list2[j])
            j += 1
    # 加入未遍历到的index
    res.extend(list1[i:]) if j == len(list2) else res.extend(list2[j:])
    return res
```

```python
def And(list1, list2):
    i, j = 0, 0
    res = []
    while i < len(list1) and j < len(list2):
        # 同时出现，加入结果列表
        if list1[i] == list2[j]:
            res.append(list1[i])
            i += 1
            j += 1
        # 指向较小数的指针后移
        elif list1[i] < list2[j]:
            i += 1
        else:
            j += 1
    return res
```

```python
def AndNot(list1, list2):
    i, j = 0, 0
    res = []
    while i < len(list1) and j < len(list2):
        # index相等时，同时后移
        if list1[i] == list2[j]:
            i += 1
            j += 1
        # 指向list1的index较小时，加入结果列表
        elif list1[i] < list2[j]:
            res.append(list1[i])
            i += 1
        else:
            j += 1
    # list1 未遍历完，加入剩余index
    if i != len(list1):
        res.extend(list1[i:])
    return res
```

由于需要分析查询效率，因此我们加入了时间函数来记录程序运行的时间，以此来分析查询效率的高低。

通过改变词项处理顺序，我们可以观察不同词项处理顺序下处理时间的变化：

```
print('美丽 AND 鲨鱼 AND 中文 AND 村上:', And(getList('中文'),And(getList('美丽'),And(getList('村上'), getList('鲨鱼')))))
```

```
程序运行时间:  0.011955261230468 75 秒
```

```
print('美丽 AND 鲨鱼 AND 中文 AND 村上:', And(getList('鲨鱼'),And(getList('村上'),And(getList('中文'), getList('美丽')))))
```

```
程序运行时间:  0.014992952346801758 秒
```

如图，通过先处理词项评率低的'鲨鱼'和'村上'，程序运行时间就会小于先处理词项频率较高的'中文'和'美丽'的运行时间。由此可见，优先处理词项频率高的词项可以提高程序的运行效率，减少运行耗时。

Section 4 索引压缩及效率提升

不同存储方式会导致查找效率的变化，本次实验中，我们设计了多种数据集倒排索引表的存储方式，包括顺序存储和按块存储等。和上个Section一样，我们通过加入时间函数来记录程序运行的时间，以此来分析查询效率的高低。

当我们使用顺序存储时，进行如下的布尔查询：

```
print('! AND !!:', And(getList('!'), getList('!!')))
print('(! AND !!)OR(2 AND 1 ):', Or(And(getList('!'), getList('!!')),And(getList('2'), getList('1'))))
print('(中文 OR 人生)ANDNOT(信息 AND 作业):', AndNot(Or((getList('中文')), getList('人生')),And(getList('信息'), getList('作业'))))
```

得到程序运行时间为：

```
程序运行时间:  0.040689945220947266 秒
```

而当我们使用倒排表按块存储后，进行同样的布尔查询，我们的程序运行时间变为了：

```
程序运行时间:  0.016998767852783203 秒
```

这是因为，在倒排表分块以后，原来n个数据的二分查找变为了五分之n个数据的二分查找，而5个数据的遍历完的时间可以几乎忽略不计，因此查找时间大大缩短了。

由此可见，不同的存储方式确实能大幅提高布尔检索及程序运行的效率。

Part 2

Section 1 数据划分

我们使用了助教提供的数据集进行数据划分。

由于数据集中的数据较为稠密，因此没有进行删减与更换。

通过修改助教提供的数据划分阶段代码，我们最终决定以 训练集：测试集 = 7 : 3 的比例划分数据。

即：train_data, test_data = train_test_split(loaded_data, test_size=0.3, random_state=42)

Section 2 基本协同过滤—基于项目

我们采取基于项目的基本协同过滤方式，预测用户对项目的评分。

（web-lab1\src_yzy\phase2\item_based_CF(basic).py）

选择基于项目而不是基于用户的推荐，是考虑到各类用户的偏好不同，而项目的属性较为单一，评分的标准相较于各人的喜好也更为一般化。基于项目的推荐相较于基于用户的推荐一般会更好。

基于项目推荐的计算公式如下：

$$r_{ix} = \frac{\Sigma_{j \epsilon N(i;x)} s_{ij} \cdot r_{jx}}{\Sigma S_{ij}}$$

其中，s_{ij}代表i、j项目的相似程度，在此我们使用简单的$Pearson$相关系数表示，使用基于同一用户进行的评分进行衡量。具体来说，就是对于i、j项目，构造两个向量，每一维代表某一个用户对该项目的评分，而两个向量的同一维来自于同一个用户的评价。

r_{jx}则代表x用户对j项目的评分。

基于上一阶段中划分出的train_data计算test_data相似度和预测值,再对结果（顺序）进行比较。

所有的计算及预测方法包含在类$item_comments$中：

```
class item_comments():

    def __init__(self, data):
        self.data = data
        self.aver = {}
        for i, (item, comment) in enumerate(self.data.items()):
            item_sum = 0.0  #评分总和
            com_sum = 0     #评分总数
            for user, value in comment.items():
                if (int(value) > -1):
                    item_sum += int(value)
                    com_sum += 1
            self.aver[item] = item_sum / com_sum

    def get_comment(self, item1):
        for item, comment in self.data.items():
            if item == item1:
                return comment
        return -1   #not found

    def pearson_sim(self, item1, item2):
```

```
        comment1 = self.get_comment(item1)
        aver1 = self.aver[item1]
        comment2 = self.get_comment(item2)
        aver2 = self.aver[item2]

        c = 0.0 #协方差
        v1 = 0.0
        v2 = 0.0    #标准差
        for item, value in comment1.items():
            for item_, value_ in comment2.items():
                if item == item_ and int(value) != -1 and int(value_) != -1 :
                    c += (float(value) - aver1) * (float(value_) - aver2)
                    v1 += pow((float(value) - aver1), 2)
                    v2 += pow((float(value_) - aver2), 2)
        v1 = pow(v1, 0.5)
        v2 = pow(v2, 0.5)

        if v1 == 0.0 or v2 == 0.0:  #没有相关项
            return 0
        else:
            return c / (v1 * v2)

    def predict_rank(self, item, user):
        numerator = 0.0     #分子
        denomintor = 0.0    #分母
        for item_, comment in self.data.items():
            if (item == item_):
                continue
            if (user in comment):
                p_s = self.pearson_sim(item, item_)
                numerator += p_s * float(comment[user])
                denomintor += p_s

        if denomintor == 0:
            return -1   #unpredictable
        else:
            return numerator/denomintor

    def solution(self):
        ret_data = self.data
        for item, comment in self.data.items():
            for user, value in comment.items():
                if int(value) == -1:
                    ret_data[item].second[user] = self.predict_rank(item, user)
        return ret_data
```

基本的文件处理不过多赘述。需要注意的是基于项目的推荐评分耗费大量的计算（在测试过程中，发现每一（用户、项目）的预测计算时间是秒级别的，于是取前2000个作为总体，参与推荐结果分析（100、2000的预测结果都存放在data文件夹当中））

Section 3 协同过滤进阶

在上文Section2部分协同过滤的基础上，我们可以使用tf-idf参数对两个item的相关性进行优化，即可以使用两文档tf-idf向量的余弦距离来取代pearson相关性
代码层面，我们需要提前预处理出每一个文档的tf-idf向量，这样在预测时可以直接进行余弦相似度的计算，在效率上能够优于之前的person相关性

```
    item_id =[]
    item_tags_list = []
    item_tags_str = []

    for i, member in enumerate(split_word):
        if(i == 0): continue
        is_pair = 0
        str = ""
        words = []
        item_tags_str.append("")
        item_tags_list.append([])

        for ch in member[1]:
            if((ch == "," or ch == '}')and is_pair == 1):
                is_pair = 0
                words.append(str[:-1])
                item_tags_str[i-1] += str[:-1] + " "
                str = ""
                continue
            if(is_pair == 1):
                str += ch
            if(ch == "'" and is_pair == 0): is_pair = 1

        id_to_order[member[0]] = i-1
        item_id.append(member[0])
        item_tags_list[i-1].append(words)

    tfidf_vec = TfidfVectorizer()

    # 使用 fit_transform() 得到 TF-IDF 矩阵
    tfidf_matrix = tfidf_vec.fit_transform(item_tags_str)
    tfidf_matrix = tfidf_matrix.toarray()

# for data format:
```

这里需要注意的一点是，tf-idf向量构成的矩阵十分稀疏，默认的储存方式是压缩过的csv_matrix，在这里我们把它直接变为nparray方便计算，当然这也导致了更大的内存占用

```
    # suppoese "item" is the id of the item
    def tf_idf_sim(self, item1, item2):
        global tfidf_matrix
        global id_to_order
        if(item1 not in id_to_order or item2 not in id_to_order):
            print("item not found in id_to_order")
            exit(0)
        order1 = id_to_order[item1]
        order2 = id_to_order[item2]
        return np.dot(tfidf_matrix[order1], tfidf_matrix[order2]) / (np.linalg.norm(tfidf_matrix[order1]) * np.linalg.norm(tfidf_matrix[order2]))

    def predict_rank(self, item, user):
```

Section 4 评价预测结果-基于 NDCG

最终我们使用 NDCG 来评价本次实验评分预测中的预测效果。

在NDCG中，我们定义累计增益：

$CG = \Sigma_R Gains$

考虑到Rank的位置，我们定义折扣累积增益：

$DCG = \Sigma_R \frac{Gains}{log_2(i+1)}$

理想情况下最大的DCG，即为降序排列时的DCG，称为IDCG

NDCG 的计算公式为：

$$NDCG = \frac{DCG}{IDCG}$$

编写程序分别计算运用基本协同过滤和协同过滤进阶的预测结果对比真实结果的NDCG，运行结果如下：

基本协同过滤：NDCG 约为0.973

进阶协同过滤：NDCG约为0.975

其中预测数据的样本量均为2000个。

NDCG的取值范围是0到1，在一些严格的竞赛或评测中，通常要求模型的NDCG值在0.8以上才能被认为是合格的。因此我们可以认为在本次实验中我们的预测结果是较为准确的。