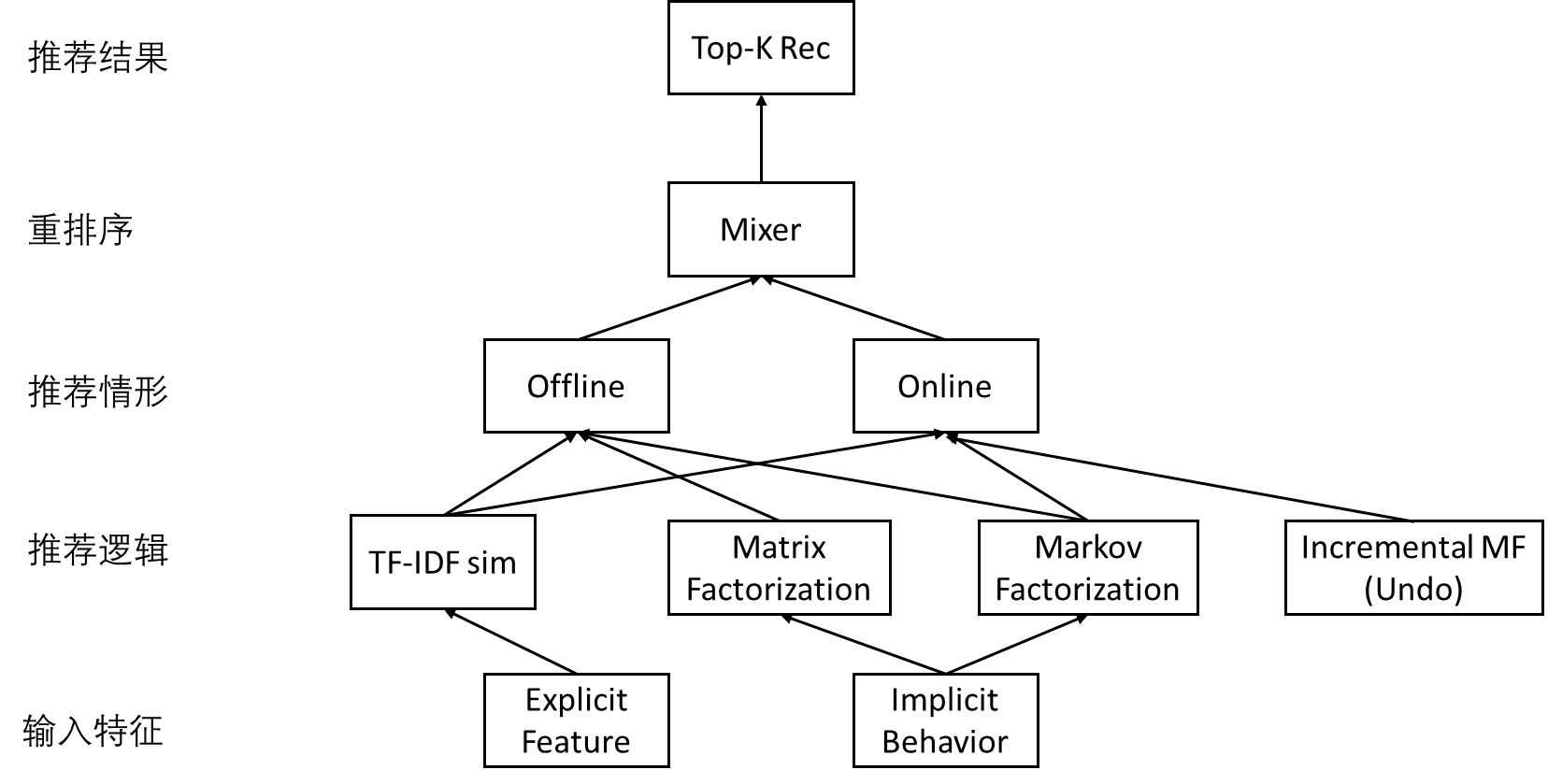
三、推荐算法

推荐算法架构：



本推荐系统的架构如上图所示：分别分为以输入特征、推荐逻辑、推荐情形、重排序、推荐结果五个部分，本文将针对这几个部分为顺序进行详细介绍，其中对应于代码也会穿插其中予以解释。该系统是利用python和spark实现的，运用了spark在python下的专业包pyspark所实现。

**3.1 输入特征**

我们要以用户的隐式行为和电影的显式特征为模型的主要数据源（Movielens2M,这个数据包含了2,000,000条用户的行为记录），对于前者我们将用户发生的行为数据视为隐式行为，因为这不是用户主动去填写的，而是系统记录下的行为用户行为数据，当今推荐系统的绝大多数工作都是以用户隐式行为数据而展开的。对于后者，可以视为电影的侧信息，是来描述电影属性（比如导演、演员等等），这些属性是电影的显式标签，所以我们用显式特征来统称这些信息。

**3.2 推荐逻辑（模型）**

3.2.1 TF-IDF相似度

对应代码：item\_item\_matrix.ipynb 文件

输入：电影的显式特征

输出：与电影i最相似的N个电影

首先我们读取电影的显式特征，将其转化为pythonRDD形式的变量，然后运用pyspark自带的TF-IDF相似度计算方法进行相似度计算，并且进行归一化处理。主要代码如下所示：

#TF-IDF计算

documents = rdd.map(lambda l: l[1].split("|"))

from pyspark.mllib.feature import HashingTF, IDF

hashingTF = HashingTF(numFeatures=1000)

tf = hashingTF.transform(documents)

tf.cache()

idf = IDF().fit(tf)

tfidf = idf.transform(tf)

#归一化处理

from pyspark.mllib.feature import Normalizer

labels = rdd.map(lambda l: l[0])

features = tfidf

normalizer = Normalizer()

data = labels.zip(normalizer.transform(features))

因为我们的item数据N非常至多，储存N^2级别的数据会消耗大量的资源，同时为了实时推荐系统的要求，从中寻找与电影i最相似的N个电影也非常耗时，于是我们希望在离线时能够快速寻找与电影i最相似的N个电影，所以我们直观地以此为返回的结果！于是我们利用python自带的稀疏矩阵去计算电影和电影的相似度，并且输入与电影i最相似的N个电影。存为文件名为item\_similarity.pkl

from scipy import sparse

import copy

coo = sparse.load\_npz('./tfidf.npz')

lil = coo.tolil()

n\_user,n\_item = lil.shape

from tqdm import tqdm

import numpy as np

import copy

batch = 100

lil\_right = lil.transpose()

topk = []

for i in tqdm(range(int(n\_user/batch))):

lil\_left = lil[i\*batch:(i+1)\*batch]

sim\_lil = lil\_left.dot(lil\_right)

v = -sim\_lil.toarray()

top5 = copy.deepcopy(np.argsort(v)[:,:5])

topk.append(top5)

3.2.2 评分矩阵分解方法

输入：用户评分矩阵

输出：用户最喜爱的N个电影

对应代码：RS.ipynb文件

对于推荐系统来说存在两大场景即评分预测与Top-N推荐。评分预测场景主要用于评价网站，比如用户给自己看过的电影评多少分（MovieLens），或者用户给自己看过的书籍评价多少分（Douban）。其中矩阵分解技术主要应用于该场景。

我们采用了pyspark自带的矩阵分解推荐算法，该方法是基于NIPS2008年的方法Probabilistic matrix factorization为损失函数，运用交替最小二乘的方法进行模型优化，主要代码具体如下所示：

————————————————————————————————————————

from pyspark.ml.recommendation import ALS

#train for model

rec = ALS(maxIter=10, regParam=0.01, userCol='userId', itemCol='movieId\_num', ratingCol='rating', nonnegative=True,

coldStartStrategy='drop')

rs\_model = rec.fit(train\_df)

我们主要返回每个用户最喜爱的N个电影座位结果，存为文件reommendation1.pkl作为离线推荐的重要依据一直之一。

3.2.3 马尔科夫矩阵分解

输入：用户评分矩阵

输出：马尔科夫相似矩阵

对应代码：markov\_sim.ipynb.ipynb文件

事实上，用户的行为是服从某种行为转移模式的，为了更好地建模用户的行为和提高推荐系统的推荐效果，我们借鉴了Rendle S在WWW2010的工作Factorizing personalized markov chains for next-basket recommendation（FPMC）来建模这种用户行为转移模式，同时为了使其适应于在线方法，我们创新性地在此基础上直接分解马尔科夫矩阵，使离线方法和在线方法都能实现。

类似于3.2.2中对于评分矩阵的分解，我们对应的分解马尔科夫转移矩阵，最终返回电影之间的马尔科夫相似矩阵，例如对于一个结果M\_i : [M\_j1,…,M\_jn]，其对应的含义为如果用户点击了M\_i，则他很有可能继续点击[M\_j1,…,M\_jn]中的电影。

3.3 推荐情形

根据推荐情形的不同，我们将其主要分为离线推荐（更新）和在线推荐（更新），前者主要以离线训练为主，旨在捕捉用户的总体偏好；后者以在线（毫秒级，绝不是离线的重新训练）策略为主，旨在捕捉用户的偏好变化。

3.3.1 离线推荐（更新）

基于用户评分矩阵的离线推荐（3.2.2），在此不赘述。

基于电影相似度的离线推荐（3.2.1和3.2.3的输出结果）

输入：电影的相似度，用户的行为

输出：用户最喜爱的N个电影

对应代码：RS2.ipynb文件和RS3.ipynb文件

该方法的主要思想就是根据用户历史行为的电影为线索，去寻找与之相似（属性相似，行为相似）的电影为依据，作为用户可能喜欢的电影，主要代码如下所示：

users\_rec = dict()

from tqdm import tqdm

import copy

import collections

values = df[['userId','movieId']].values

recurrent\_list = []

for i,line in tqdm(enumerate(values)):

user,item = line

append = copy.deepcopy(item\_simpickle[item].tolist())

if values[i][0] == values[i+1][0]:

recurrent\_list.extend(append)

else:

Counter = collections.Counter(recurrent\_list)

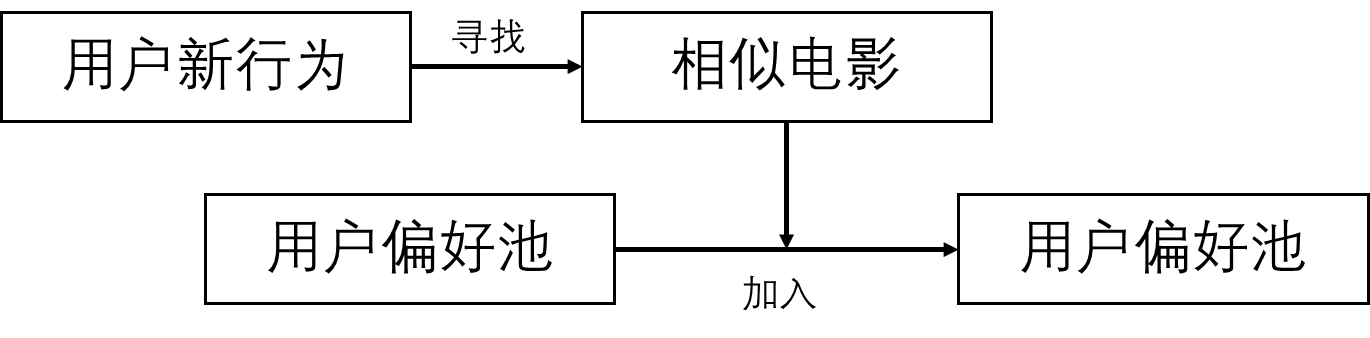
rec\_user = copy.deepcopy(list(Counter.keys())[::-1][:10])

users\_rec[user]= rec\_user

recurrent\_list = []

其中运用Counter去寻找出现次数最多的电影，最终返回用户在不同相似度下可能喜爱的TopK个电影，结果如recommendation2和recommendation3所示。

3.3.2 在线推荐

因为用户希望实时的与系统交互，而不是第二天或者等待数分钟后发现当前行为的结果，所以我们希望构建一种毫秒级用户偏好池，以此为依据进行推荐。直观的，更新逻辑如下所示：

显然相似电影是我们在3.2.1和3.2.3中所构建好的，只需O(1)去查询即可，然后需要O(1)去加入用户偏好池，于是整个过程在理论上是O(1)时间复杂度的，可以满足用户的实时性要求。为了实现实时性，在后端实现推荐，在此不列。

5.4 重排序

因为本组人员较少，重排序只是作为一个抽象模块出现，我们简化的具体实现是一个叠加的过程，随机打乱结果，根据上述所有的推荐逻辑返回的结果进行推荐。

5.5 Top-K推荐

根据重排序后的结果推荐前N个用户可能喜爱的电影。