11 | 召回层：如何快速又准确地筛选掉不相关物品？

你好，我是王喆。今天，我们来一起学习推荐系统中非常重要的一个模块，召回层。

为了弄清楚召回层是什么，我们先试着解决一下这个问题：**如果你是一名快手的推荐工程师，你的任务是从 500 万个候选短视频中，为一名用户推荐 10 个他最感兴趣的。你会怎么做？**

我想最直接最暴力的做法，就是对这 500 万个短视频挨个打分、排序，取出得分最高的 10 个推荐给用户。如果这个打分的算法非常靠谱的话，我们肯定能够选出用户最感兴趣的 Top 10 视频。但这个过程会涉及一个非常棘手的工程问题：如果利用比较复杂的推荐模型，特别是深度学习推荐模型，对 500 万个短视频打分，这个过程是非常消耗计算资源的。

而且你要知道，这还只是计算了一个用户的推荐结果，在工业级的线上服务中，每秒可是有几十万甚至上百万的用户同时请求服务器，逐个候选视频打分产生的计算量，是任何集群都承受不了的。

那**在推荐物品候选集规模非常大的时候，我们该如何快速又准确地筛选掉不相关物品，从而节约排序时所消耗的计算资源呢？**这其实就是推荐系统召回层要解决的问题。今天，我就从三个召回层技术方案入手，带你一起来解决这个问题。

召回层和排序层的功能特点

在前面的课程中我提到过学习推荐系统的一个主要原则，那就是“深入细节，不忘整体”。对于召回层，我们也应该清楚它在推荐系统架构中的位置。

从技术架构的角度来说，“召回层”处于推荐系统的**线上服务模块**之中，推荐服务器从数据库或内存中拿到所有候选物品集合后，会依次经过召回层、排序层、再排序层（也被称为补充算法层），才能够产生用户最终看到的推荐列表。既然线上服务需要这么多“层”才能产生最终的结果，不同层之间的功能特点有什么区别呢？

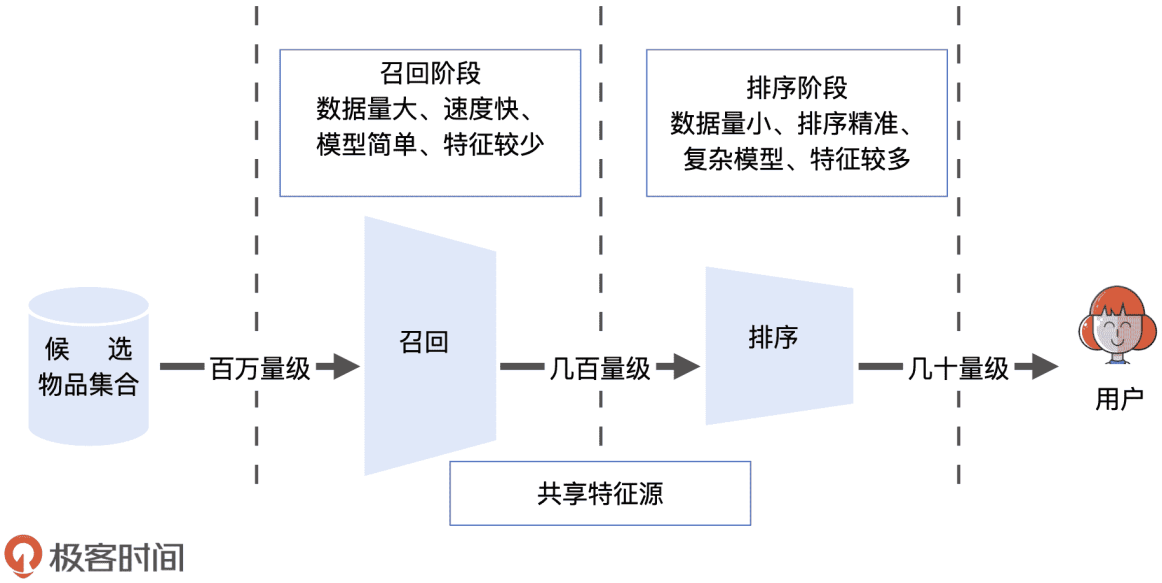

图1 推荐系统的召回和排序阶段及其特点

其实从这节课开头的问题出发，你应该已经对召回层和排序层的功能特点有了初步的认识，召回层就是要**快速**、准确地过滤出相关物品，缩小候选集，排序层则要以提升推荐效果为目，作出精准的推荐列表排序。

再详细一点说，我们可以从候选集规模、模型复杂程度、特征数量、处理速度、排序精度等几个角度来对比召回层和排序层的特点：

	候选集规模	模型复杂度	所用特征	处理速度	排序精度
召回层	大（百万量级）	低	少	非常快	较低
排序层	小（几百量级）	高	多	较慢	高

图2 召回层和排序层的特点

需要注意的是，在我们设计召回层时，计算速度和召回率其实是两个矛盾的指标。怎么理解呢？比如说，为了提高计算速度，我们需要使召回策略尽量简单，而为了提高召回率或者说召回精度，则要求召回策略能够尽量把用户感兴趣的物品囊括在内，这又要求召回策略不能过于简单，导致召回物品无法满足排序模型的要求。

推荐工程师们就是在这样的矛盾中逐渐作出新的尝试，推动着召回层的设计方案不断向前发展。下面，我们就详细学习一下三个主要的召回方法，以及它们基于 SparrowRecSys 的代码实现。

如何理解“单策略召回”方法？

你会发现，今天我多次提到一个关键字，快。那怎么才能让召回层“快”起来呢？我们知道，召回层慢的原因是模型复杂，算法计算量大，那我们能不能反其道而行之，用一些简单直观的策略来实现召回层呢？当然是可以的，这就是所谓的**单策略召回**。

单策略召回指的是，通过制定一条规则或者利用一个简单模型来快速地召回可能的相关物品。 这里的规则其实就是用户可能感兴趣的物品

的特点，我们拿 SparrowRecSys 里面的电影推荐为例。在推荐电影的时候，我们首先要想到用户可能会喜欢什么电影。按照经验来说，很有可能是这三类，分别是大众口碑好的、近期非常火热的，以及跟我之前喜欢的电影风格类似的。

基于其中任何一条，我们都可以快速实现一个单策略召回层。比如在 SparrowRecSys 中，我就制定了这样一条召回策略：如果用户对电影 A 的评分较高，比如超过 4 分，那么我们就将与 A 风格相同，并且平均评分在前 50 的电影召回，放入排序候选集中。

基于这条规则，我实现了如下的召回层：

```
//详见SimilarMovieFlow class
public static List<Movie> candidateGenerator(Movie
    ArrayList<Movie> candidates = new ArrayList<>(
    //使用HashMap去重
    HashMap<Integer, Movie> candidateMap = new Has
    //电影movie包含多个风格标签
    for (String genre : movie.getGenres()){

        //召回策略的实现
        List<Movie> oneCandidates = DataManager.ge
        for (Movie candidate : oneCandidates){
            candidateMap.put(candidate.getMovieId(
        }
    }
    //去掉movie本身
    if (candidateMap.containsKey(movie.getMovieId(
        candidateMap.remove(movie.getMovieId());
    }
    //最终的候选集
    return new ArrayList<>(candidateMap.values());
}
```

单策略召回是非常简单直观的，正因为简单，所以它的计算速度一定是非常快的。但我想你应该也发现了其中的问题，就是它有很强的局限性。因为大多数时候用户的兴趣是非常多元的，他们不仅喜欢自己感兴趣的，也喜欢热门的，当然很多时候也喜欢新上映的。这时候，单一策略就难以满足用户的潜在需求了，那有没有更全面的召回策略呢？

如何理解“多路召回”方法

为了让召回的结果更加全面，多路召回方法应运而生了。

所谓“多路召回策略”，就是指采用不同的策略、特征或简单模型，分别召回一部分候选集，然后把候选集混合在一起供后续排序模型使用的策略。

其中，各简单策略保证候选集的快速召回，从不同角度设计的策略又能保证召回率接近理想的状态，不至于损害排序效果。所以，多路召回策略是在计算速度和召回率之间进行权衡的结果。

这里，我们还是以电影推荐为例来做进一步的解释。下面是我给出的电影推荐中常用的多路召回策略，包括热门电影、风格类型、高分评价、最新上映以及朋友喜欢等等。除此之外，我们也可以把一些推断速度比较快的简单模型（比如逻辑回归，协同过滤等）生成的推荐结果放入多路召回层中，形成综合性更好的候选集。具体的操作过程就是，我们分别执行这些策略，让每个策略选取 Top k 个物品，最后混合多个 Top k 物品，就形成了最终的多路召回候选集。整个过程就如下所示：

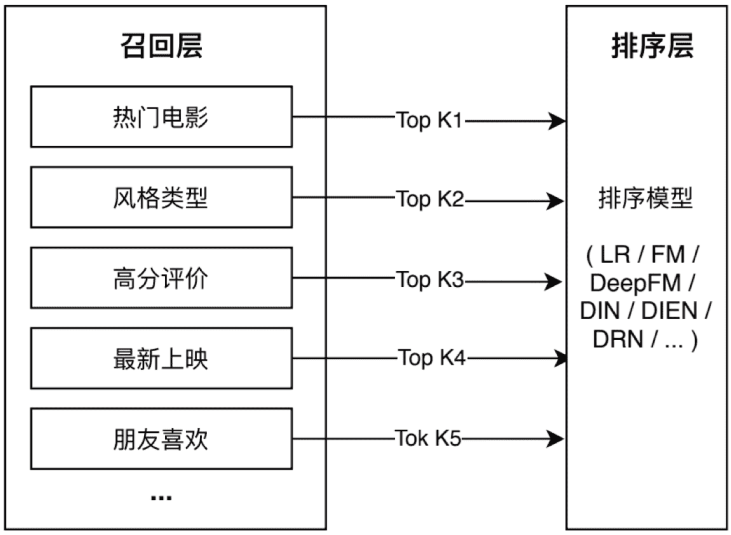

图3 常见的多路召回策略

在 SparrowRecsys 中，我们就实现了由风格类型、高分评价、最新上映，这三路召回策略组成的多路召回方法，具体代码如下：

```
public static List<Movie> multipleRetrievalCandida
    HashSet<String> genres = new HashSet<>();
    //根据用户看过的电影，统计用户喜欢的电影风格
    for (Movie movie : userHistory){
        genres.addAll(movie.getGenres());
    }
    //根据用户喜欢的风格召回电影候选集
    HashMap<Integer, Movie> candidateMap = new Has
    for (String genre : genres){
        List<Movie> oneCandidates = DataManager.ge
        for (Movie candidate : oneCandidates){
            candidateMap.put(candidate.getMovieId(
        }
    }
    //召回所有电影中排名最高的100部电影
    List<Movie> highRatingCandidates = DataManager
```

```
    List<Movie> highRatingCandidates = DataManager
    for (Movie candidate : highRatingCandidates){
        candidateMap.put(candidate.getMovieId(), c
    }
    //召回最新上映的100部电影
    List<Movie> latestCandidates = DataManager.get
    for (Movie candidate : latestCandidates){
        candidateMap.put(candidate.getMovieId(), c
    }
    //去除用户已经观看过的电影
    for (Movie movie : userHistory){
        candidateMap.remove(movie.getMovieId());
    }
    //形成最终的候选集
    return new ArrayList<>(candidateMap.values());
}
```

在实现的过程中，为了进一步优化召回效率，我们还可以通过多线程并行、建立标签 / 特征索引、建立常用召回集缓存等方法来进一步完善它。

不过，多路召回策略虽然能够比较全面地照顾到不同的召回方法，但也存在一些缺点。比如，在确定每一路的召回物品数量时，往往需要大量的人工参与和调整，具体的数值需要经过大量线上 AB 测试来决定。此外，因为策略之间的信息和数据是割裂的，所以我们很难综合考虑不同策略对一个物品的影响。

那么，是否存在一个综合性强且计算速度也能满足需求的召回方法呢？

基于 Embedding 的召回方法

在第 5 节和第 6 节课中，我们已经介绍了多种离线生成物品 Embedding 的方案。事实上，利用物品和用户 Embedding 相似性来构建召回层，是深度学习推荐系统中非常经典的技术方案。我们可以把它的优势总结为三方面。

一方面，多路召回中使用的“兴趣标签”“热门度”“流行趋势”“物品属性”等信息都可以作为 Embedding 方法中的附加信息（Side information），融合进最终的 Embedding 向量中 。因此，在利用 Embedding 召回的过程中，我们就相当于考虑到了多路召回的多种策略。

另一方面，Embedding 召回的评分具有连续性。我们知道，多路召回中不同召回策略产生的相似度、热度等分值不具备可比性，所以我们无法据此来决定每个召回策略放回候选集的大小。但是，Embedding 召回却可以把 Embedding 间的相似度作为唯一的判断标准，因此它可以随意限定召回的候选集大小。

最后，在线上服务的过程中，Embedding 相似性的计算也相对简单和直接。通过简单的点积或余弦相似度的运算就能够得到相似度得分，便于线上的快速召回。

在 SparrowRecsys 中，我们也实现了基于 Embedding 的召回方法。我具体代码放在下面，你可以参考一下。

```
public static List<Movie> retrievalCandidatesByEmb
    if (null == user){
        return null;
    }
    //获取用户embedding向量
    double[] userEmbedding = DataManager.getInstan
    if (null == userEmbedding){
```

```
            return null;
        }
        //获取所有影片候选集(这里取评分排名前10000的影片作为全
        List<Movie> allCandidates = DataManager.getIns
        HashMap<Movie,Double> movieScoreMap = new Hash
        //逐一获取电影embedding，并计算与用户embedding的相似
        for (Movie candidate : allCandidates){

            double[] itemEmbedding = DataManager.getIn
            double similarity = calculateEmbeddingSimi
            movieScoreMap.put(candidate, similarity);
        }

        List<Map.Entry<Movie,Double>> movieScoreList =
        //按照用户-电影embedding相似度进行候选电影集排序
        movieScoreList.sort(Map.Entry.comparingByValue


        //生成并返回最终的候选集
        List<Movie> candidates = new ArrayList<>();
        for (Map.Entry<Movie,Double> movieScoreEntry :
            candidates.add(movieScoreEntry.getKey());
        }
        return candidates.subList(0, Math.min(candidat
    }
```

这里，我再带你简单梳理一下整体的实现思路。总的来说，我们通过 3 步生成了最终的候选集。第一步，我们获取用户的 Embedding。第二步，我们获取所有物品的候选集，并且逐一获取物品的 Embedding，计算物品 Embedding 和用户 Embedding 的相似度。第三步，我们根据相似度排序，返回规定大小的候选集。

在这 3 步之中，最主要的时间开销在第 2 步，虽然它的时间复杂度是线性的，但当物品集过大时（比如达到了百万以上的规模），线性的运算也可能造成很大的时间开销。那有没有什么方法能进一步缩小 Embedding 召回层的运算时间呢？这个问题我们留到下节课来讨论。

小结

今天，我们一起讨论了推荐系统中召回层的功能特点和实现方法。并且重点讲解了单策略召回、多路召回，以及深度学习推荐系统中常用的基于 Embedding 的召回。

为了方便你对比它们之间的技术特点，我总结了一张表格放在了下面，你可以看一看。

召回方法	主要特点	优势	局限性
单策略召回	利用评分、标签、新鲜度等单一策略召回用户可能喜欢的物品	速度快、实现简单	策略简单， 无法覆盖不同召回需求， 可能漏掉大量感兴趣物品
多路召回	融合多个单一策略的召回方案	可以覆盖不同的召回需求	需要大量人工调参 来决定每一路的规模， 每一路的得分不具备可比性
Embedding召回	基于预训练好的用户、物品Embedding向量实现的召回策略	可以在Embedding预训练中融合大量用户和物品特征， 线上部分评分连续， 实现也较简单	线上计算部分还需进一步优化

极客时间

总的来说，关于召回层的重要内容，我总结成了**一个特点，三个方案**。

特点就是召回层的功能特点：召回层要快速准确地过滤出相关物品，缩小候选集。三个方案指的是实现召回层的三个技术方案：简单快速

的单策略召回、业界主流的多路召回、深度学习推荐系统中最常用的 Embedding 召回。

这三种方法基本囊括了现在业界推荐系统的主流召回方法，希望通过这节课的学习，你能掌握这一关键模块的实现方法。

相信你也一定发现了，召回层技术的发展是循序渐进的，因此我希望你不仅能够学会应用它们，更能够站在前人的技术基础上，进一步推进它的发展，这也是工程师这份职业最大的魅力。

课后思考

1. 你能根据我今天讲的内容在 SparrowRecsys 中实现一个多线程版本的多路召回策略吗？
2. 你觉得对于 Embedding 召回来说，我们怎么做才能提升计算 Embedding 相似度的速度？

你理解的召回层也是这样吗？欢迎把你的思考和答案写在留言区。如果有收获，我也希望你能把这节课分享给你的朋友们。