浙江财经大学

本 科 生 毕 业 论 文（设计）

题目：基于分布式混合推荐的商品推荐系统：融合在线、近线和离线

学生姓名 朱姚林

学　　号 190110950639

指导教师 季家兵

所在学院 数据科学学院

专业名称 数据科学与大数据技术

班　　级 19 大数据

2023 年 5 月

基于分布式混合推荐的商品推荐系统：融合在线、近线和离线

摘要：推荐系统已成为解决信息过载问题的重要工具，在电商、社交网络、影视等领域都有广泛应用。本文实现了一个基于分布式计算和存储的商品推荐系统，该系统参考了 Netflix 发布的推荐系统三层架构，分为在线层、近线层和离线层，各个层级采用不同的计算和存储框架，以满足不同处理的时效性要求：离线层负责训练机器学习模型、计算相似度索引等大规模任务，近线层读取实时数据并通过流式处理框架进行处理，在线层负责与用户直接交互并捕捉用户行为。在各个层级上，穿插实现了包括内容过滤、协同过滤、矩阵分解在内的多种算法。通过在系统架构和模型算法的混合，推荐系统可以更准确、更高效地推荐商品，增强用户体验。

关键词：推荐系统；混合推荐；分布式计算

A Distributed Hybrid Product Recommendation System: Integration of Online, Near-line, and Offline

Abstract: Recommendation system has become an important tool to solve the problem of information overload, and has been widely used in e-commerce, social networks, movies and other fields. This paper implements a commodity recommendation system based on distributed computing and storage. The system refers to the three-layer architecture of the recommendation system released by Netflix, which is divided into online layer, near-line layer, and offline layer. Different computing and storage frameworks are used in each layer to meet the time efficiency requirements of different processing. The offline layer is responsible for training machine learning models and calculating similarity indexes and other large-scale tasks. The near-line layer reads real-time data and processes it through stream processing frameworks, while the online layer is responsible for directly interacting with users and capturing user behavior. At each level, various algorithms including content filtering, collaborative filtering and matrix decomposition are implemented. Through the combination of system architecture and model algorithms, the recommendation system can recommend products more accurately and efficiently, and enhance the user experience.

Key words: Recommendation System; Hybrid Recommendation; Distributed Computing

目 录

1 引 言

自上世纪 90 年代起，推荐系统就一直是一个重要的研究领域。作为解决信息过载问题的重要工具[1]，推荐系统通过分析用户的行为和偏好，向他们推荐可能感兴趣的商品、服务或内容。推荐系统旨在提高用户体验和满意度，从而促进用户忠诚度和销售额的增长。推荐系统在电子商务、社交网络、影视电影等领域都有广泛应用[2]。其中，电子商务领域的推荐系统发展最为成熟，成功的经典案例包括 Amazon、淘宝等电商巨头；在影视电影领域，推荐系统可以向用户推荐他们喜欢的影视剧和电影，成功的经典案例包括 Netflix[3]、爱奇艺等影视巨头。

推荐算法是推荐系统的核心，它通过协同过滤、内容过滤、机器学习等技术，从海量的用户数据中提取出有用的信息，为用户提供个性化的推荐服务。基于协同过滤的算法通过分析用户历史行为和其他用户的行为，找出用户之间的相似性，从而给用户推荐感兴趣的物品。基于内容过滤的算法则是基于物品的属性或标签等信息，给用户推荐与他们过去兴趣相似的物品。而基于机器学习的算法则可以对大量的数据进行自动化学习，从而实现更加准确和精细的推荐。

对于一个工业化的推荐系统，推荐的质量和效率都同等重要。为了实现高效和准确的推荐服务，大数据技术成为了推荐系统中必不可少的组成部分。各种分布式计算框架，如 Spark，可以高效处理海量数据，实现高性能的数据分析和处理。凭借分布式计算的能力，推荐系统可以更加高效地进行推荐计算和模型训练，提高系统的推荐效率和准确性。流处理技术也是推荐系统中重要的技术之一，该技术可以处理实时的数据流，从而实现对用户行为的实时分析和推荐服务。例如，推荐系统可以实时监测用户在网站上的行为，分析用户行为模式和偏好，并基于这些信息实时调整推荐结果，提高用户的满意度和体验。

本文将从推荐算法理论和具体实现的两个角度出发，实现一个分布式的商品推荐系统。论文的第二章将介绍推荐系统的算法和用到的大数据技术，并介绍推荐系统的三层架构，第三章将对商品推荐系统的架构和算法进行设计，第四章将对商品推荐系统进行具体的实现，第五章对系统进行了测试和联调，第六章是全文的总结和展望。

2 经典推荐系统算法及架构

本章将介绍本文采用的推荐算法、本文和 Netflix 采用的推荐系统框架（重叠的部分将只介绍本文采用的技术，如 Redis 和 EVCache 将只介绍本文采用的 Redis）；随后以数据流的方式介绍 Netflix 推荐系统的三层架构。

2.1 常见推荐算法

2.1.1 内容过滤

内容过滤 (Content-based filtering, CBF) 是一种基于物品属性的推荐算法，它通过分析物品的特征向量，找到相似性最高的物品并推荐给用户[4]。该算法不需要了解用户的行为和偏好，只依赖于用户过去喜欢的物品的属性完成推荐。

物品的向量化是 CBF 的关键步骤。常见的方法是使用 TF-IDF (Term Frequency-Inverse Document Frequency) 方法对文本类型的物品进行向量化。具体地，TF-IDF 方法将一个文本类型的物品表示为一个向量，向量的每个分量对应一个词项，分量的取值为 TF 与 IDF 的乘积，表示该词项在文本中出现的频率和重要程度。其中，TF 表示词项在文本中的出现频率，IDF 表示词项在文本集合中的逆文档频率，二者的计算公式为：

$$\begin{aligned} TF_{i,j} &= \frac{n_{i,j}}{\sum_k n_{k,j}}, \\ IDF_i &= \log \frac{|D|}{1 + \left|j : t_i \in d_j\right|}. \end{aligned} \tag{2.1}$$

其中，$n_{i,j}$ 表示词语 t_i 在文档 d_j 中出现的次数，$\sum_k n_{k,j}$ 表示文档 d_j 的总词语数，$|D|$ 表示所有文档的数量，$\left|j : t_i \in d_j\right|$ 表示包含词语 t_i 的文档数量。随后计算物品的相似度，设 V_a 表示物品 a 的特征向量，V_b 表示物品 b 的特征向量，两个物品的余弦相似度定义为：

$$sim(a,b) = \cos\langle V_a, V_b \rangle = \frac{V_a \cdot V_b}{\|V_a\| \|V_b\|}. \tag{2.2}$$

最后将与用户喜欢的物品相似度最高的 k 个物品推荐给用户。CBF 最典型的应用场景是在用户对一个物品作出好评或点击后，在其页面中推荐与其相似的物品。

2.1.2 基于物品的协同过滤

基于物品的协同过滤 (Item-based Collaborative Filtering, ItemCF) 是一种基于物品相似度和用户历史行为的推荐算法，它通过用户对已交互过的物品的兴趣以及物品之间的相似度来预估用户对候选物品的兴趣，找到兴趣得分最高的候选物品并推荐给用户[5]。

ItemCF 的核心思想是：如果用户喜欢物品 a，且物品 a 与物品 b 相似，则用户也可能会喜欢物品 b。ItemCF 的相似度计算一般不仅仅考虑物品的属性，而是着重考虑用户的行为，常见的方法是使用 Ochiia 系数，设 A 为喜欢物品 a 的用户集合，B 为喜欢物品 b 的用户集合，用两个集合的交集大小和两个集合大小的几何平均数的比值

作为两个物品的相似度：

$$sim(a,b) = \frac{|A \cap B|}{\sqrt{|A| \times |B|}}. \tag{2.3}$$

计算完两两物品之间的相似度之后就可以计算用户对候选物品的兴趣。若某一用户交互过的物品集合为 W，那么用户对候选物品 c 的兴趣为：

$$like(c) = \sum\nolimits_{w \in W} like(w) \times sim(w,c). \tag{2.4}$$

其中 $like(x)$ 代表用户对物品 x 的兴趣。ItemCF 计算用户对所有候选物品的兴趣，选择兴趣最高的物品并推荐给用户。通常来说，候选物品集合 C 与用户交互过的物品集合 W 互为补集。

2.1.3 基于用户的协同过滤

基于用户的协同过滤 (User-based Collaborative Filtering, UserCF) 是一种基于用户相似度和用户历史行为的推荐算法，它通过用户对已交互过的物品的兴趣以及用户之间的相似度来预估用户对候选物品的兴趣，找到兴趣得分最高的候选物品并推荐给用户[6]。

UserCF 的核心思想是：如果用户 a 和用户 b 相似，且用户 b 喜欢某一件候选物品，那么用户 a 也可能会喜欢这一物品[7]。UserCF 的相似度计算也采用 Ochiia 系数，设 A 为用户 a 喜欢的物品集合，B 为用户 b 喜欢的物品集合，那么两个用户的相似度为：

$$sim(a,b) = \frac{|A \cap B|}{\sqrt{|A| \times |B|}}. \tag{2.5}$$

随后计算用户对候选物品的兴趣。记交互过候选物品的用户集合为 W，则用户 c 对候选物品的兴趣为：

$$like(c) = \sum\nolimits_{w \in W} sim(c,w) \times like(w)\,, \tag{2.6}$$

其中 $like(x)$ 代表用户 x 对物品的兴趣。UserCF 计算用户对所有候选物品的兴趣，选择兴趣最高的物品并推荐给用户。通常来说，候选物品集合与用户交互过的物品集合互为补集。

2.1.4 矩阵补全和矩阵分解

矩阵补全 (Matrix Completion) 是推荐系统中的一个重要问题。记某一用户对商品的评分矩阵 $R = (r_{ij})_{m \times n}$，矩阵元素 r_{ij} $(r_{ij} \geq 0 \text{ or } r_{ij} = nan)$ 代表用户 i 对物品 j 的评分，推荐系统需要根据该矩阵为用户推荐评分最高且用户没有查看过的物品。然而，

矩阵 R 中的大多数评分都是缺失的，因此矩阵补全的任务就是要把一个含有大量缺失值的矩阵通过一定的方法恢复成一个完整的矩阵。矩阵补全问题包含一个重要的假设，即用户的偏好是相似的（或物品的特性是相似的），该假设决定了矩阵 R 是低秩矩阵，因此可以借助矩阵中冗余的信息对缺失值进行填补。

矩阵分解 (Matrix Factorization) 是用于求解矩阵补全问题的主流方法[8]，对于评分矩阵 R，将其分解为用户特征矩阵 $U=(U_1,U_2,\cdots,U_m)=(u_{ij})_{k\times m}$，以及物品特征矩阵 $V=(V_1,V_2,\cdots,V_n)=(v_{ij})_{k\times n}$，使得两个矩阵的乘积 $U^{\mathrm{T}}V$ 尽可能接近原始矩阵 R，如图 2.1 所示。其中，U_i 和 V_j 分别是用户特征向量和物品特征向量，二者的内积就是用户 i 对物品 j 的预测评分 $\hat{r}_{ij}$，k 是可调整的隐含特征的维度。具体的，令 $\widehat{R}=U^{\mathrm{T}}V$，最小化评分矩阵和乘积矩阵的欧式距离的平方，那么目标函数为：

$$\begin{aligned}\min_{U,V} J &= \left\|R-\widehat{R}\right\|^2 = \|R-U^{\mathrm{T}}V\|^2 \\ &= \sum_{i,j}\left(r_{ij}-U_i^{\mathrm{T}}V_j\right)^2 \\ &= \sum_{i,j}\left(r_{ij}-\sum\nolimits_{l=1}^{k}u_{li}v_{lj}\right)^2,\end{aligned} \tag{2.7}$$

但是，这样的优化目标会涉及到缺失值的插补，为了避免错误的缺失值插补导致的严重精度问题，需要考虑去除掉缺失值的模型。另外，为了避免过拟合，同时为目标加入 L2 正则项[9]：

$$\begin{aligned}\min_{U,V} J &= \sum_{\substack{i,j \\ r_{ij}\neq nan}}\left(r_{ij}-U_i^{\mathrm{T}}V_j\right)^2+\lambda\sum_i\|U_i\|^2+\lambda\sum_j\|V_j\|^2 \\ &= \sum_{\substack{i,j \\ r_{ij}\neq nan}}\left(r_{ij}-\sum\nolimits_{l=1}^{k}u_{li}v_{lj}\right)^2+\sum_{l,i}u_{li}^2+\sum_{l,j}v_{lj}^2 .\end{aligned} \tag{2.8}$$

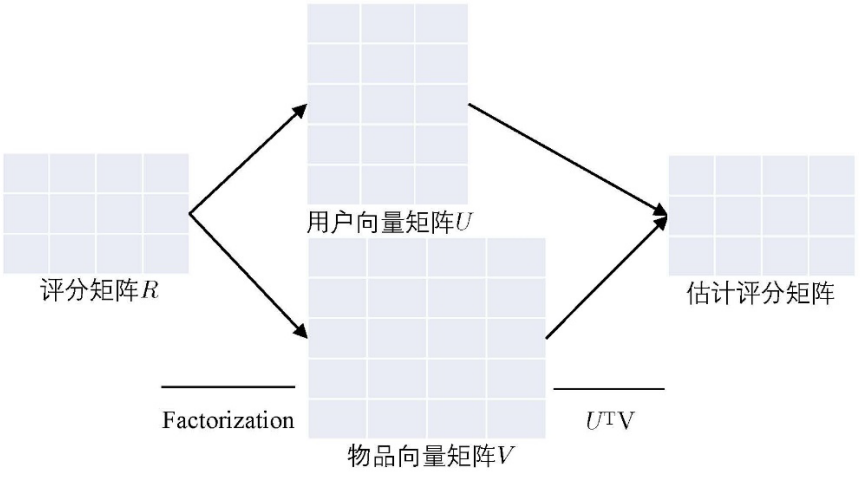

图 2.1　将评分矩阵分解为用户矩阵和物品矩阵

2.1.5 分布式交替最小二乘法

由于 $U_i^{\mathrm{T}}V_j = \sum_{l=1}^{k} u_{li}v_{lj}$ 的存在，(2.8) 式中的目标函数 J 是非凸的，事实上优化该目标函数是一个 NP-hard 的问题。随机梯度下降法可以用于优化该目标函数，但计算效率低下，需要大量的迭代。但是，若将用户矩阵 U 固定并视作常数，那么目标就是关于用户矩阵 V 的凸函数，反之亦然。这种计算的方法被称为交替最小二乘法 (Alternating Least Squares, ALS)。以固定矩阵 V 从而优化矩阵 U 为例，对于某一个具体的优化参数 U_i，其目标函数可以更改为：

$$J_{U_i} = \sum_{\substack{j \\ r_{ij} \neq nan}} (r_{ij} - U_i^{\mathrm{T}}V_j)^2 + \lambda \|U_i\|^2 . \tag{2.9}$$

计算函数对 U_i 的偏导数，并令其等于 0：

$$\frac{\partial J_{U_i}}{\partial U_i} = - \sum_{\substack{j \\ r_{ij} \neq nan}} 2(r_{ij} - U_i^{\mathrm{T}}V_j)V_j + 2\lambda U_i = 0. \tag{2.10}$$

通过用户特征向量和物品特征向量的含义可知，$U_i^{\mathrm{T}}V_j = V_j^{\mathrm{T}}U_i = \widehat{R}_{ij}$ 是一个常数，因此与 V_j 交换位置该式的结果不变。化简 (2.10) 式可得：

$$\begin{aligned}
&\sum_{\substack{j \\ r_{ij} \neq nan}} V_j(V_j^{\mathrm{T}}U_i) - \sum_{\substack{j \\ r_{ij} \neq nan}} r_{ij}V_j + \lambda U_i = 0, \\
\Rightarrow &\sum_{\substack{j \\ r_{ij} \neq nan}} (V_jV_j^{\mathrm{T}} + \lambda I_k)U_i = \sum_{\substack{j \\ r_{ij} \neq nan}} r_{ij}V_j, \\
\Rightarrow &\, r_{ij} \neq nan\text{: } (VV^{\mathrm{T}} + \lambda I_k)U_i = V(R^{\mathrm{T}})_i, \\
\Rightarrow &\, r_{ij} \neq nan\text{: } U_i = (VV^{\mathrm{T}} + \lambda I_k)^{-1}V(R^{\mathrm{T}})_i,
\end{aligned} \tag{2.11}$$

其中 $(R^{\mathrm{T}})_i$ 表示评分矩阵的转置的第 i 列向量。对于每一个用户向量，均执行上述更新。随后固定矩阵 U 优化矩阵 V ，重复循环该步骤直到收敛。ALS 用于矩阵分解的全部步骤如下：

repeat
  for $i = 1 \ldots m$ **do**

$$r_{ij} \neq nan\text{: } U_i = (VV^{\mathrm{T}} + \lambda I_k)^{-1}V(R^{\mathrm{T}})_i \tag{2.12}$$

  end for
  for $j = 1 \ldots n$ **do**

$$r_{ij} \neq nan\text{: } V_j = (UU^{\mathrm{T}} + \lambda I_k)^{-1}UR_i \tag{2.13}$$

  end for
until *Convergence*

可以计算单节点 ALS 算法的计算复杂度，每次更新 U_i 需要花费 $O(n_i k^2 + k^3)$，每次更新 U_j 需要花费 $O(n_j k^2 + k^3)$。其中 n_i 是用户 i 的评分数量，n_j 是评价过物品 j 的用户数量。对于一个复杂的推荐系统，需要引入更多的节点用于训练模型，同时也需要适用于分布式环境的 ALS 算法。对于一个完全分布式的环境，所有的数据和参数都是分布式的，在本文的实验环境下，它们都以 Spark 中的 RDD 形式存在[10]。例如，评分矩阵 R 被存储为三元组形式的 RDD，这是因为 R 一般是稀疏矩阵：

$$R : \mathrm{RDD}[(i, j, r_{ij}), \dots]. \tag{2.14}$$

用户矩阵 U 和物品矩阵 V 并不是稀疏矩阵，因此它们被存储为向量形式的 RDD：

$$U : \mathrm{RDD}[U_1, \dots, U_m], \tag{2.15}$$

$$V : \mathrm{RDD}[V_1, \dots, V_m]. \tag{2.16}$$

以 (2.12) 式为例，分布式计算的重点在于 VV^{T} 以及 $V(R^{\mathrm{T}})_i$，它们在 Spark 集群中的计算步骤如下[11]：

1) 将 R 和 V 以物品 j 为 key 创建 pairRDD，并 R join V 得到 joinedRDD。
2) joinedRDD 用 map 计算 $V_j V_j^{\mathrm{T}}$ ，并将 key 从物品 j 转变成用户 i。
3) 以用户 i 为 key 执行 reduceByKey，累加 $V_j V_j^{\mathrm{T}}$ 得到 VV^{T}。
4) joinedRDD 用 map 计算 $r_{ij} V_j$。
5) 以用户 i 为 key 执行 reduceByKey，累加 $r_{ij} V_j$ 得到 $V(R^{\mathrm{T}})_i$。

其中 join、map 和 reduceByKey 分别满足[10]：

1) $join(): (\mathrm{RDD}[(\mathrm{K}, \mathrm{V})], \mathrm{RDD}[(\mathrm{K}, \mathrm{W})]) \Rightarrow \mathrm{RDD}[(\mathrm{K}, (\mathrm{V}, \mathrm{W}))]$
2) $map(f: \mathrm{T} \Rightarrow \mathrm{U}): \mathrm{RDD}[\mathrm{T}] \Rightarrow \mathrm{RDD}[\mathrm{U}]$
3) $reduceByKey(f: (\mathrm{V}, \mathrm{V}) \Rightarrow \mathrm{V}): R\mathrm{DD}[(\mathrm{K}, \mathrm{V})] \Rightarrow \mathrm{RDD}[(\mathrm{K}, \mathrm{V})]$

2.2 常见系统框架

2.2.1 计算框架

- MapReduce

  Hadoop MapReduce 是一种用于大规模数据处理的编程模型和算法。它最初由 Google 于 2004 年提出，后来被 Apache Hadoop 项目采纳并实现。MapReduce 将数据处理任务分解为两个基本阶段：Map 阶段和 Reduce 阶段。

  在 Map 阶段，数据被切分成多个块，并由 Map 函数对每个块进行处理。Map 函数将输入数据转换为一组键值对，并将这些键值对传递给 Reduce 函数进行处理。中间经过 Shuffle 后进入 Reduce 阶段，Reduce 函数将具有相同键的键值对进行聚合，

并生成最终的输出结果[12]。

MapReduce 的优点在于它的可扩展性和容错性。由于 MapReduce 可以在大量计算机上并行运行，因此可以轻松地处理非常大的数据集。此外，MapReduce 也具有高容错性，即使在一个或多个节点出现故障时，MapReduce 仍然可以继续进行数据处理。

尽管 MapReduce 是一种强大的工具，但它也存在一些缺点。例如，MapReduce 的编程模型相对较低级别，需要开发人员编写大量的代码来执行简单的任务。此外，由于 MapReduce 的磁盘 I/O 开销较大，因此处理速度相对较慢。

- Apache Spark

Apache Spark 是一种快速、通用、可扩展的大数据处理引擎，它支持在分布式环境中进行内存计算。Spark 最初是由加州大学伯克利分校 AMP 实验室于 2009 年开发的，后来成为 Apache 软件基金会的一个顶级开源项目。Spark 为开发人员提供了一种分布式数据处理的简单方式，并且比 Hadoop MapReduce 更快、更易用。Spark 虽然代替不了 Haodop 生态圈，但是能完美地代替 MapReduce。

Spark 采用基于 JVM 的 Scala 开发，并支持各种编程语言，包括 Java、Scala、Python 和 R 等，并且提供了高层次的 API，如 Spark SQL[13]、Spark Streaming、MLlib 和 GraphX 等。Spark 提供了一种分布式的内存计算模型，称为 Resilient Distributed Datasets (RDDs)。RDD 是一个可分区的、可伸缩的、容错的数据结构，它允许在大规模数据集上进行高效的并行计算。

- Apache Flink

Apache Flink 是一个开源的流处理框架，它提供了高效、可扩展的数据流处理技术，可以在实时数据流和批量数据流之间无缝切换。Flink 最初由德国柏林技术大学于 2010 年开始开发，后来成为 Apache 软件基金会的一个顶级开源项目。

Flink 使用 Java 和 Scala 语言进行开发，提供了丰富的 API 和工具包，包括 DataStream API、DataSet API、Table API、CEP（复杂事件处理）API 和 ML（机器学习）库等。Flink 的数据流处理模型非常灵活，可以同时处理批量和实时数据，并支持对窗口、时间和状态进行灵活的控制。

Flink 通过流式计算来实现数据流处理，这种计算方式可以保证实时性，同时具有高吞吐量和低延迟的优势。Flink 的计算模型是基于有向无环图 (DAG) 的，数据流通过多个算子进行转换，最终形成一个计算图[14]。Flink 提供了多种内存管理机制和数据分区策略，可以让用户灵活地控制计算资源的使用。

2.2.2 存储框架

- HDFS

  Apache Hadoop Distributed File System (HDFS) 是一种用于存储和处理大规模数据的分布式文件系统。HDFS 最初是由 Apache Hadoop 项目开发的，现在是 Apache 软件基金会的一个顶级开源项目之一。

  HDFS 是一个基于 Java 编写的文件系统，具有高可靠性、高可扩展性和高吞吐量的特点。它将大文件划分成小块，并在多个计算节点上进行存储和处理。HDFS 还使用了多个副本来保证数据的可靠性，并能够自动将副本分布在不同的节点上，以提高数据可用性[15]。

  HDFS 架构由两个核心组件组成：NameNode 和 DataNode。NameNode 是 HDFS 的主要控制节点，负责管理文件系统命名空间、访问控制和副本策略等。DataNode 则是存储实际数据的节点，负责读写文件块和执行副本复制等操作。HDFS 是一个可靠、高扩展性和高吞吐量的分布式文件系统，适用于存储和处理大规模数据。

- Redis

  Redis 是一种开源的基于内存的键值存储系统，旨在提供高性能和可靠的数据存储。Redis 最初由 Salvatore Sanfilippo 开发，它采用了基于键值的数据模型，支持多种数据结构，如字符串、哈希表、列表和集合。Redis 能够存储和检索大量的数据，并支持高吞吐量和低延迟的数据访问，使其成为处理实时数据和分布式键值数据库的理想解决方案。

  Redis 的数据模型是基于内存的，这使得它具有高速的读写性能。它还支持磁盘持久化，可以将数据写入磁盘以保证数据的安全性和可靠性。Redis 具有高度的可扩展性和灵活性，可以通过集群来扩展存储容量和吞吐量。Redis 还支持多种数据复制模式，包括主从复制和集群模式，以提供高可用性和数据冗余。

- MongoDB

  MongoDB 是一个流行的文档数据库，它以高性能、易扩展和灵活的数据模型而闻名。MongoDB 是一个开源项目，由 MongoDB 公司维护。它可以与多个编程语言和开发框架一起使用，包括 Java、Python、Node.js 和 Scala 等。

  MongoDB 使用面向文档的数据模型，这意味着数据以类似 JSON 的 BSON 格式存储在一个文档中，而不是以行和列的形式存储。这种数据模型使得 MongoDB 可以轻松处理半结构化数据和非常复杂的数据。

  MongoDB 是一个分布式数据库系统，它支持自动分片，这意味着数据可以水平分割到多个服务器上，从而支持高可用性和可扩展性。MongoDB 还提供了副本集，

这是一组数据副本，可以提供数据冗余和故障转移功能。副本集中的每个副本都可以接收读请求，并且自动进行故障转移。

2.2.3 调度框架和中间件

- Flume

  Apache Flume 是一个开源的分布式日志收集、聚合和传输系统，由 Apache 软件基金会进行维护和管理。它支持可靠的、高可用性的流式数据传输，可以用于实时数据处理、日志采集和数据挖掘等领域。Flume 的核心理念是将数据从源头传输到目的地，其中数据源可以是文件、日志、网络流等，而目的地可以是 Hadoop HDFS、Kafka、HBase 等数据存储和处理系统。

  Flume 由多个组件组成，其中最常用的组件是 Source、Channel 和 Sink。Source 负责从数据源获取数据，并将数据传输到 Channel；Channel 存储数据，并将其传输到 Sink；Sink 负责将数据从 Channel 传输到目的地。

  Flume 的优点是高可靠性、高可扩展性和灵活性。它可以通过配置文件和插件来实现各种数据传输场景，并且支持故障转移和数据重传等机制，保证数据传输的可靠性和完整性。同时，Flume 的架构可以根据需要进行扩展和定制，以适应不同的数据处理需求。

- Kafka

  Apache Kafka 是一个分布式流处理平台，由 Apache 软件基金会维护和开发。Kafka 是一个高吞吐量、低延迟的平台，可以处理大规模的实时数据。它主要用于处理和传输大量的数据流，例如日志和事件数据等。

  Kafka 使用发布/订阅模型来管理数据流，可以将消息发布到一个或多个主题(topics)，然后消费者可以订阅一个或多个主题以接收消息。Kafka 的关键设计是将数据流存储在分布式集群中的多个节点上，并提供可扩展性和容错性。它还提供了许多功能，如数据压缩、消息持久性、数据分区、数据副本等[16]。

  Kafka 支持多种编程语言，包括 Java、Python、Scala 等，并且具有广泛的社区支持。它还与许多其他工具和框架（如 Spark、Flink、Hadoop 等）集成，可以用于构建可靠的、实时的数据处理和分析系统。

- Zookeeper

  Zookeeper 是一个开源的分布式协调服务，由 Apache 软件基金会维护和开发。它主要用于协调和管理分布式应用程序中的各种元数据，如配置信息、命名服务、状态信息等。Zookeeper 采用了一个层次结构的命名空间来存储这些元数据，并提供

了高可用性、高性能和可扩展性。

Zookeeper 的设计目标是提供一个简单而强大的分布式锁服务，以便其他应用程序可以使用它来实现各种分布式协作模式，例如选举、协调和同步。Zookeeper 的关键特性包括原子性、可靠性、顺序性和持久性。它提供了丰富的 API，可以用于构建可靠的、高性能的分布式应用程序。

Zookeeper 与 Apache Kafka 密切相关，因为 Kafka 使用 Zookeeper 来管理主题和分区的元数据，并协调生产者和消费者之间的数据交换。Zookeeper 还可以用于其他分布式应用程序中，例如 Hadoop、HBase、Storm 等。Zookeeper 具有广泛的社区支持，可以在多个操作系统和编程语言中使用。

2.3 Netflix 推荐系统架构

Netflix 将他们的推荐系统架构分为三层，这三层分别是：在线层 (Online)，近线层 (Nearline) 和离线层 (Offline)，整个系统部署在亚马逊云平台 AWS 上，如图 2.2 所示。以在线层的用户为起点，整个系统可以分为两条链路，一条通往近线层，一条通往离线层，最终二者再回归到在线层。在线层是直接面向用户的，用户通过 UI 客户端和系统进行直接交互，将行为直接传递给系统，而系统则反馈推荐列表给用户。用户的行为通过打点日志的方式收集起来，然后通过事件分布 (Event Distribution) 将事件发送到不同的分布式节点上，包括离线层的 Hadoop 和近线层的消息队列，即上文提到的两条链路。

对于通往离线层的链路：数据会先存储在 Hadoop 的 HDFS 中，这类数据往往会以天粒度或小时粒度，采用 Hive 或 Pig 进行加工处理，这一处理包括数据的过滤和提取。查询的结果将会传给 Netflix.Hermes，Netflix.Herms 是一个由 Netflix 开发的分布式消息传递平台，用于推荐系统中的异步通信解决方案。Netflix 获取查询结果以及历史数据并进行三个分发：模型训练、离线计算和回馈在线层，模型训练部分会使用到一些复杂的机器学习和深度学习的算法，例如矩阵分解和双塔模型，训练好的模型参数以及一些训练过程中产生的副产品（例如矩阵分解过程中产生的物品向量和用户向量）也会打回到在线层进行使用。

对于通往近线层的链路：数据会以消息队列的形式从事件分布中获取数据，随后传递给 Netflix.Manhattan，Netflix.Manhattn 是一个流式处理平台，数据通过流式处理平台触发近线计算，近线计算除了获取流式数据，还可能会从在线层的数据服务中获取数据。数据通过一些简单的机器学习算法（如协同过滤）计算推荐结果和中间结果，并将数据写入到 Cassandra、MySQL 和 EVCache。其中，Cassandra 是一个 NoSQL 的高性能分布式数据库，用于海量数据的存储并提供高速的读取服务；EVCache 是一种分布式缓存的解决方案，将数据存储在分布式内存中，提供更加高效的读取服务，进一步提高系统的

吞吐量；而 MySQL 则用于为系统提供结构化的查询操作，但是其查询和写入的速度相对较低，因此仅起到辅助的作用。

两条链路回到在线层，在线层直接调用其他层的服务，例如读取近线层中数据库保存的内容；在线层也可以进行一些简单的计算，例如采用离线层模型进行预测。最终的结果统一通过算法服务的形式将推荐列表返回给 UI 客户端，最终呈现给用户。

Netflix 的推荐系统架构不仅为用户提供了个性化和高质量的推荐服务；同时也具备良好的系统性能和扩展性，能够支持更多用户和更多数据量的处理；为用户带来了良好的观影体验。随着时间的发展，整套系统中的诸多技术也在不断地迭代更新，但 Netflix 的框架架构因其简洁清晰的分层和运行逻辑，至今仍然是推荐系统中的主流。

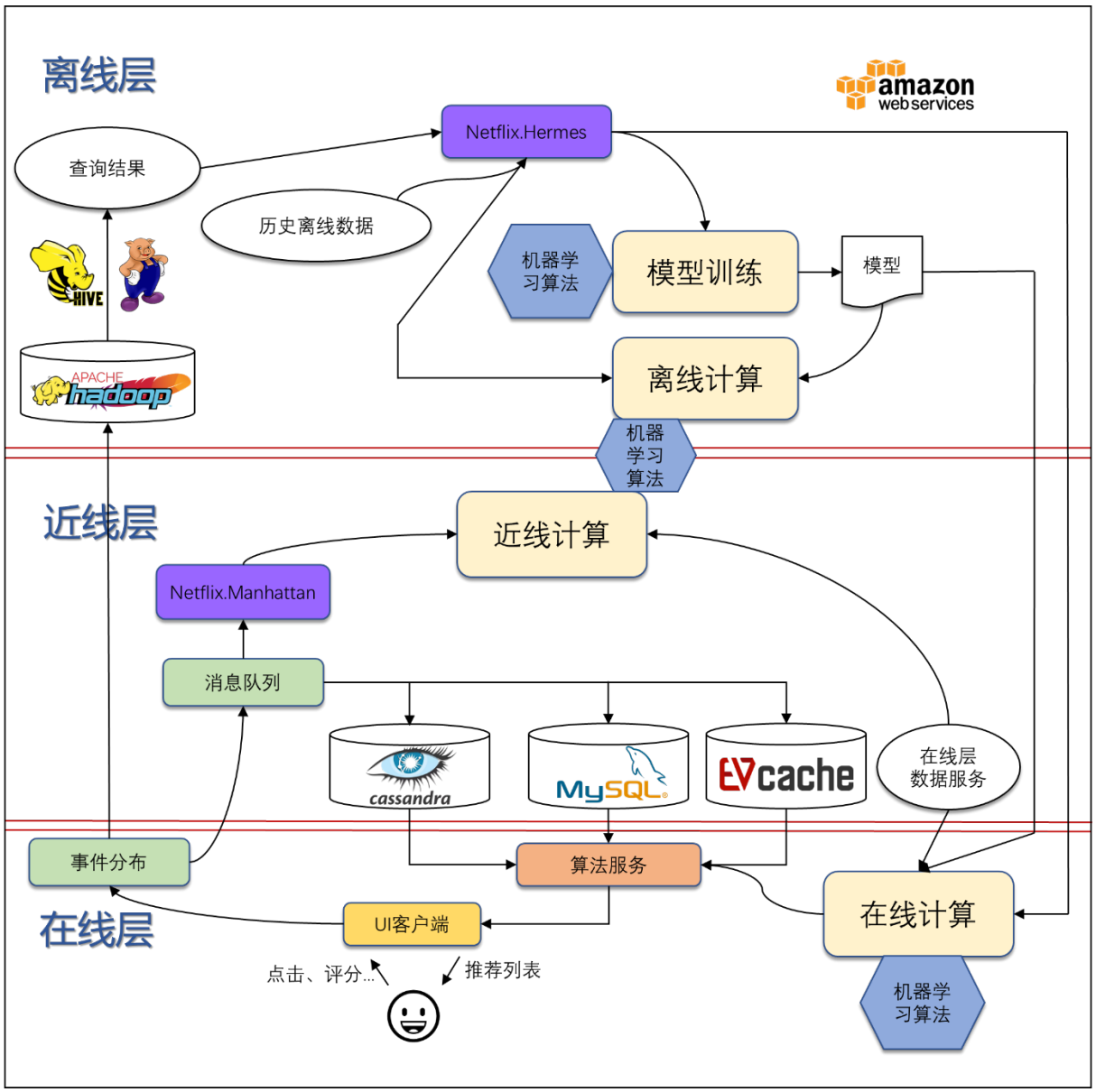

图 2.2　Netflix 推荐系统架构

3　商品推荐系统架构和算法设计

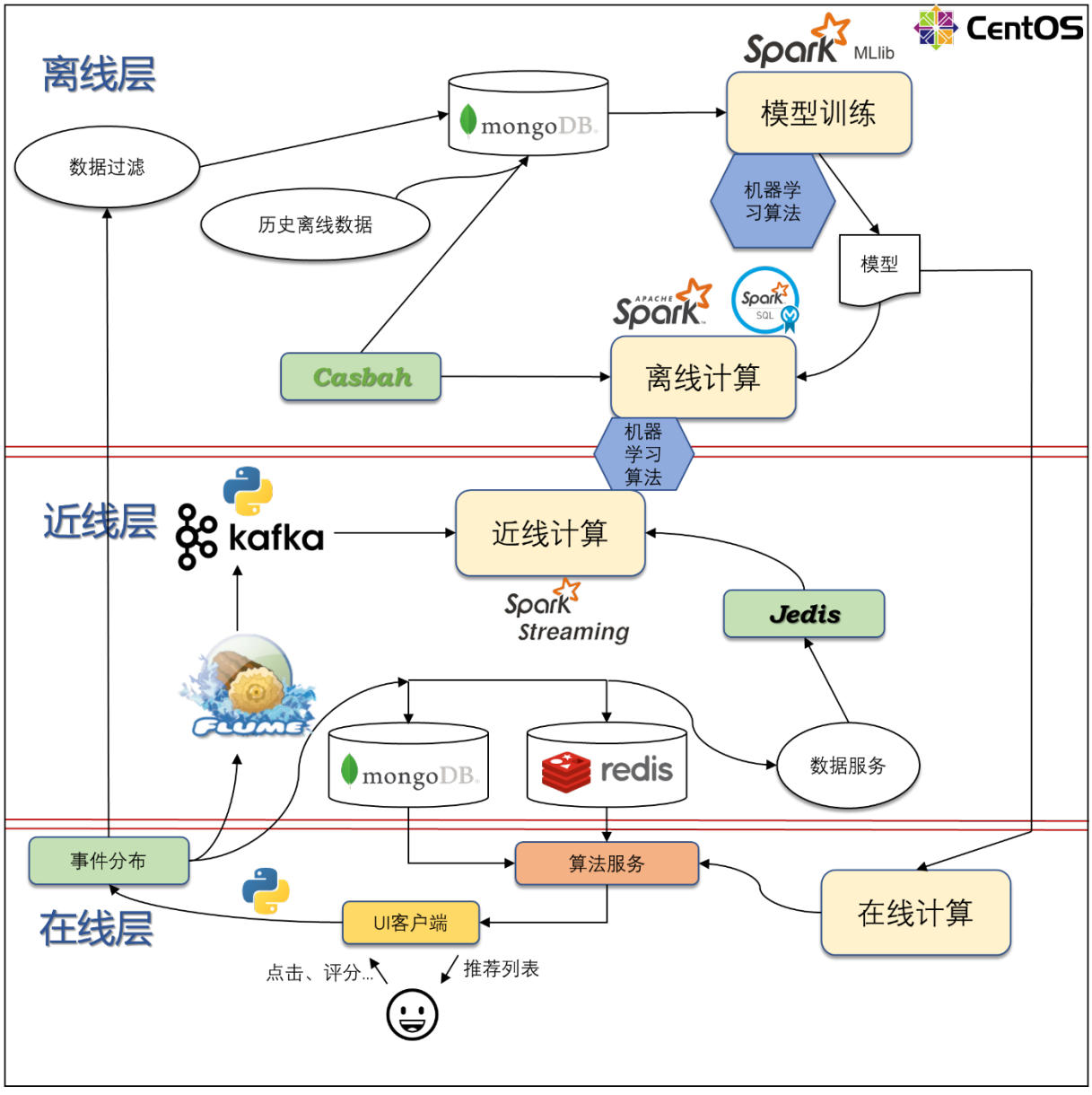

图 3.1　商品推荐系统架构设计

商品推荐系统架构同样分为在线层、近线层和离线层，整个系统部署在分布式环境下，如图 3.1 所示。以在线层的用户为起点，整个系统依然可以分为两条链路，一条通往近线层，一条通往离线层，最终再回归到在线层。整体架构与 Netflix 推荐系统架构类似，数据流略有不同，

对于通往离线层的链路：打点日志数据通过数据分布，一部分数据经过整理存储在 MongoDB 中，结合历史的离线数据，MongoDB 中保存着用户对商品的评分数据、商品信息数据；另一部分会传递到 Flume 和 Redis。存储在 MongoDB 中的数据，其中一部

分用于离线的机器学习训练，例如训练矩阵分解模型，另一部分用于普通的离线计算，得到的计算结果（推荐列表）都将写回 MongoDB 供在线层调用，同时离线计算可以计算物品相似度索引和用户相似度索引，供近线层调用。

对于通往近线层的链路，数据将传入日志采集系统，由日志采集系统写入消息队列，对于消息队列中的日志数据，进行模式匹配和清洗供近线计算调用。Spark Streaming 获取消息队列中的数据，同时分别借助 Jedis 和 Casbah 从近线层数据服务中获取辅助数据（即相似度索引），结合两者计算用户的推荐列表并写回 MongoDB。MongoDB 和 Redis 的多种存储为近线层提供了高速、大吞吐量的数据存取服务。

两条链路最终回到在线层，在线层从 MongoDB 中获取数据，从算法服务中获取推荐列表，将推荐列表返回给客户端，呈现给用户。三个层级之间互相配合，实现分布式混合推荐。

3.1 离线层

离线层是商品推荐系统的核心组件，负责处理大量数据，根据用户历史行为和其他因素，建立起推荐系统所需的模型。在这个层级中，使用了一些批处理的算法和模型，如显示反馈的矩阵分解模型和协同过滤等技术，利用分布式计算框架 Apache Spark 来提高计算性能和数据处理能力。离线层的数据处理和模型训练需要更长的时间，实时性较差，无法做到推荐结果和中间数据的实时更新，但离线层可以完成更高质量的推荐，并提前准备部分数据结构供近线层和在线层调用。推荐系统的离线层功能最多，设计最为复杂，在本文的实现中有如下几个重要的模块：

- 数据导入模块

  在实际的推荐系统中，数据导入模块可能会涉及从 HDFS 或数据仓库中，采用 Hive、Sqoop 等工具将数据导入到业务数据库中。在商品推荐系统中，数据以文本文件的形式存储在本地磁盘，因此只需要通过 Spark SQL 将数据导入到 MongoDB 中，使 MongoDB 中存在用户商品评分集合和商品信息集合。

- 统计推荐模块

  统计推荐模块在一般的推荐系统中都有实现，该模块设计简单，推荐效果也能符合实际的热点，同时，统计推荐还不会受到冷启动问题的影响。在商品推荐系统中，统计推荐模块会从 MongoDB 中读取用户商品评分集合，随后通过 Spark SQL 做出热门推荐、优质推荐等，将推荐结果写回业务数据库，供在线层调用。统计推荐模块的推荐列表是非个性化的，因此所有用户都会得到相同的推荐结果。

- 内容过滤模块

  在实际的推荐系统中，内容过滤模块通常会和其他模块联动设计，内容过滤算法可以通过自然语言处理的方式计算物品之间的相似度，甚至可以考虑用户对物品的文本评论。在商品推荐系统中，内容过滤模块会读取商品信息集合，获取商品的用户标签信息 (Tags)，采用 Spark ML 提供的 TF 和 IDF 工具计算商品向量，计算各商品之间的余弦相似度，并按照相似度的倒序将结果写回业务数据库，供在线层调用。内容过滤模块的推荐列表是针对商品的，即为喜欢某一特定商品的用户推荐与其相似的商品。

- 协同过滤模块

  协同过滤算法是最著名的推荐算法，不同于内容过滤和基于模型的算法，协同过滤算法是基于近邻和用户行为的算法，该算法通过分析用户历史行为和偏好，发现用户或物品之间的相似性，并根据这种相似性进行推荐。协同过滤包括 Item-CF 和 User-CF，由于在近线层的实时推荐模块中将实现 Item-CF 的变体，因此在协同过滤模块中仅实现 User-CF。该模块读取用户商品评分集合，为每个用户计算对出所有物品的偏好，按照数组的形式将推荐列表保存到业务数据库，供在线层调用。

- 矩阵分解模块

  矩阵分解算法因其功能的强大，在实际的推荐系统中被大量应用，它的主要思想是将用户-物品评分矩阵分解为两个较小的矩阵，从而得到用户和物品的低维向量表示。这些向量表示具有很强的语义信息，可以用于计算用户对物品的喜好度，从而实现推荐功能。在商品推荐系统的矩阵分解模块中，从 MongoDB 中读取用户物品评分矩阵，并将其划分为训练集合验证集，采用网格搜索交叉验证的方法最小化 RMSE 以寻找最优参数，并采用 Spark Mllib 提供的 ALS 算法进行模型的训练。另外，矩阵分解模块会计算物品之间的相似度索引，该相似度索引不同于内容过滤模块，还考虑了用户行为之间的相关性，拥有更强的指导意义，该相似度索引会和推荐结果一起存入业务数据库，供在线层和近线层调用。

3.2 近线层

近线层是一个介于在线层和离线层之间的层级，它提供快速的查询和响应。近线层可以快速响应在线层的请求，其数据处理时间通常比离线层更长，但比在线层短。近线层通常会使用一些缓存技术，如 Redis 和 Memcached，用以提高查询和响应的性能。同时，为了近线层的扩展性和可伸缩性，近线层还会使用一些大规模的高性能的 NoSQL 数据库，例如 HBase、MongoDB 和 Cassandra[17]。近线层会通过流式处理平台获取消息

队列中的数据，以分钟级或秒级的能力处理数据[18]。实际上，随着各种分布式计算框架和计算机软硬件的发展，近线层也逐渐拥有了一定的批处理能力，近线层和离线层之间界限被逐渐打破。

近线层计算的触发较为特殊，不同于离线层的时间触发机制和在线层的请求触发机制，近线层计算依靠事件触发，即从事件分布得到事件，在具体的实现中多为实时流处理系统从消息队列中获取了一条数据。

在商品推荐系统中，近线层在实时推荐模块中实现。在线层产生的日志数据（即用户的评价数据）会被 Flume 采集并传递到 Kafka 的 log topic，通过 Kafka-Python 对日志进行模式匹配和过滤，过滤后的数据存于 Kafka 的 recommend topic。随后流式处理平台 Spark Streaming 将消费日志数据并触发一次响应，从 Redis 和 MongoDB 中获取辅助数据，并计算最新的推荐列表。

3.3 在线层

在线层是推荐系统中的前后端组件，用于处理用户实时地请求并显示推荐结果，在线层需要快速响应请求，因此需要使用高可用、分布式、低延迟的系统进行实现，例如配备多个数据中心和负载均衡。在线层在实际使用中通常会调用下层提供的服务，在极端情况下，在线层不做任何额外的计算，直接从缓存或业务数据库中提取结果并返回给用户。领域特定语言 (Domain Specific Language, DSL) 是推荐系统逻辑架构设计中常用的一种描述，下面是一个 DSL 描述在线层的可能案例，该在线层通过调用下层的服务进行推荐结果的更新：

```
module = GoodsRecommender
recallers = MatrixFactorization + ItemCF + UserCF
matrixFactorizationHost = Node1: 1111
ItemCFHost = Node1: 2222
UserCFHost = Node2: 1111
```

在这个案例中，在线层通过调用其他节点上的矩阵分解模型、ItemCF 和 UserCF 计算出的结果来更新推荐列表。在本文实现的系统中，在线层不包括前后端的处理逻辑，而是利用程序脚本的形式模拟用户浏览网站的行为，模拟打点日志采集。最终的推荐列表会存入业务数据库 MongoDB，在线层可以在此基础上对推荐结果进行融合和重排。

4 商品推荐系统实现

4.1 数据导入模块

本文采用亚马逊评价数据集 (Amazon - Ratings)，该数据集记录了亚马逊用户对网站中商品的评分信息，该数据集共包含两张表，分别是商品信息表和用户评分表，二者的字段和示例信息如表 4.1 所示。

表 4.1 商品信息和用户评分表的字段和示例

商品信息表		用户评分表	
字段	示例	字段	示例
商品 ID	122417	用户 ID	535648
商品名称	不能承受的生命之轻	商品 ID	122417
商品分类 ID	832,723,396	评分	5.0
亚马逊 ID	B003YYSFRY	时间戳	1322236800
图片链接	\		
商品分类信息	哲学|人文社科|图书		
商品标签	小说|米兰昆德拉|好看		

将无用的字段过滤并导入到 MongoDB 中，每张表都对应着数据库中的一个集合 (Colletion)，其中商品信息表对应于集合 Products，用户评分表对应于集合 Ratings，每条记录对应着集合中的一条文档 (Document)，表 4.2 展示了文档的字段和示例信息。

表 4.2 Products 和 Ratings 集合的字段和示例

Products		Ratings	
字段	示例	字段	示例
_id	643faa5c157e0c3b745df3b4	_id	643faa5d157e0c3b745e4868
productID	122417	userID	535648
productName	不能承受的生命之轻	productID	122417
categories	哲学|人文社科|图书	score	5.0
tags	小说|米兰昆德拉|好看	timestamp	1322236800

4.2 统计推荐模块

统计推荐模块用于完成商品的热门商品推荐、近期热门推荐和优质商品推荐，属于离线的非个性化推荐，建议将运行的时间颗粒度设置为天。该模块从 MongoDB 中读取 Ratings 集合，将其类型转换为 org.apache.spark.sql.DataFrame，创建临时视图 ratings，随后采用 Spark SQL 获得推荐列表并写回 MongoDB。

热门商品推荐以 productId 作为分组字段，计算 ratings 中每件商品评分的数量，将结果列表按照评分数量的降序写回数据库。近期热门推荐通过注册 UDF 函数，将时间

戳转换为 yyyyMM 的标准日期格式，随后按照日期和 productId 字段进行分组，计算出 ratings 中每件商品评分的数量，最后将结果列表按照日期和评分数量的降序写回数据库。优质商品推荐以 productId 作为分组字段，计算 ratings 中每件商品的平均评分，最后将结果列表按照平均评分的降序写回数据库。三者分别可以按照下面的示例计算：

```
ratings.createOrReplaceTempview("ratings")
spark.udf.register("transform", (x: Int) =>
new SimpleDateFormat("yyyyMM").format(new Date(x * 1000L)).toInt))
spark.sql("""
  select productId, count(productId) as count from ratings
  group by productId
  order by count desc """)

spark.sql("""
  select productId, count(productId) as count, transform(timestamp) as date
  from ratings
  group by date, productId
  order by date desc, count desc """)

spark.sql("""
  select productId, avg(score) as avg from ratings
  group by productId
  order by avg desc """)
```

4.3 内容过滤模块

内容过滤模块用于完成商品的相似度推荐，且相似度的计算不依赖于用户行为数据，只依赖于物品的文本数据。若商品的文本数据不变或没有商品的增加，则内容过滤模块仅需要运行一遍。内容过滤模块最终将推荐列表（即倒序的商品相似度列表）存入 MongoDB，供在线层调用。

内容过滤模块从 MongoDB 中读取 Products 集合，提取商品的 tags 标签，利用 Mllib 中的文本特征处理模块训练 TF-IDF 模型。通过 Tokenizer 将数据集中的标签 (tags) 列转换为分词列并使用 HashingTF 和 IDF 计算出 TF-IDF 作为物品特征向量。

```
val tokenizer = new Tokenizer().setInputCol("tags").setOutputCol("words")
val hashingTF = new HashingTF().setInputCol("words").setOutputCol("tf")
hashingTF = hashingTF.setNumFeatures(300)
val idf = new IDF().setInputCol("tf").setOutputCol("tfidf")

val splitData = tokenizer.transform(data)
```

```
val tfData = hashingTF.transform(splitData)
val idfModel = idf.fit(tfData)
val tfIdfData = idfModel.transform(tfData)
```

最后计算两两物品向量之间的余弦相似度，按照相似度的倒序进行排序并存入MongoDB，即可作为内容过滤模块的推荐列表。

4.4 协同过滤模块

在实时推荐模块，本文会实现一种类似于 Item-CF 的商品推荐算法，该算法会利用到离线层计算完毕的物品相似度索引。而本模块将实现 User-CF，并且采用 (2.5) 式计算用户相似度索引。由于相似度索引计算较为耗时，且候选物品为所有物品，因此计算量远大于实时推荐模块，建议将协同过滤模块的运行时间颗粒度设置为分钟级别。

协同过滤模块会首先计算出用户之间的相似度索引，将 userRDD：RDD[userId] 转化为 userIndex：RDD[userId, Seq((userId, similarity),…)]，随后计算各个用户对候选物品的偏好，并按照偏好的降序写回 MongoDB 数据库：

```
val candidate = productRDD.distinct()
val userIndex = similarUsers(userRDD, ratings)
val predict = score(candidate, userIndex, ratings)
val result = predict
  .filter(_.rating > 0)
  .map(x => (x.user, (x.product, x.rating)))
  .groupByKey()
  .map {
    case (user, proRat) =>
      (user, proRat.toList.sortWith(_._2 > _._2).take(MAX_NUM))
  }
```

在协同过滤模块，候选物品集合被直接选择为所有物品，并没有对用户已经浏览过的物品进行剔除，这是因为在大多数推荐系统中，“去已读”这一步骤是在重排阶段完成的（实际上重排阶段还会完成包括去重、打散、多样性在内的多项工作），但是本文实现的所有模块都属于召回阶段，因此不需要完成这一步骤。

4.5 矩阵分解模块

矩阵分解模块用于完成商品的个性化离线推荐，同时会计算出各商品的相似度索引，将其持久化存储至 MongoDB，用于实时推荐模块。由于涉及到大量的数据训练，建议将矩阵分解模块的运行时间颗粒度设置为天或更高。该模块从 MongoDB 中读取 Ratings

集合，将其划分为训练集合验证集，利用验证集测试适用于 ALS 算法的参数，选择最优参数并训练矩阵分解模型并计算商品相似度索引，最后将推荐结果矩阵和相似度索引写回 MongoDB，用于在线层和近线层的调用。

从 MongoDB 中读取出 Ratings 后，过滤掉 timestamp 字段，并将其类型转换为适用于 MLlib 包的 RDD[org.apache.spark.mllib.recommendation.Rating]（或者适用于 ML 包的 org.apache.spark.sql.DataFrame），并将数据按照 8:2 的比例划分为训练集和验证集。由于超参数的数量量较少，因此可以通过网格搜索 (GridSearchCV) 选择最优参数，包括隐含特征的维度 k 和正则化系数 λ。具体的，设定迭代次数为 10，k 的搜索范围设定为 5 到 25，λ 的搜索范围设定为 $(5, 1, 0.3, 0.1, 0.01)$，以上述参数训练 ALS 模型并计算均方根误差 RMSE，选取 RMSE 最小的参数形成元组作为函数的返回：

```
def GridSearchCV (trainData: RDD[Rating], validationData: RDD[Rating]):
(Int, Double) = {
  val result = for (rank <- 5 to 25; lambda <- Seq(5, 1, 0.3, 0.1, 0.01))
  yield {
    val model = ALS.train(trainData, rank, iterations, lambda)
    val rmse = RMSE(model, validationData)
    (rank, lambda, rmse)
  }
  val optimalParameters = result.minBy(_._3)
  (optimalParameters._1, optimalParameters._2)
}
```

矩阵分解模型的 RMSE 与传统机器学习算法的 RMSE 略有不同，需要略过用户没有对物品进行过评分的项：

$$RMSE = \sqrt{\frac{1}{N_{obs}} \sum_{i,j,r_{ij} \neq nan} (r_{ij} - \hat{r}_{ij})^2}, \tag{4.1}$$

其中 N_{obs} 表示矩阵 R 中所有非缺失值的数量。略过缺失值的 RMSE 可以通过 join 实现，以 (i, j) 为 key，通过 join 连接预测评分和真实评分，随后采用模式匹配即可只选择非缺失值进行计算：

```
def RMSE(model: MatrixFactorizationModel, data: RDD[Rating]): Double = {
  val index = data.map(item => (item.user, item.product))
  val predict = model.predict(index).map(
                      item => ((item.user, item.product), item.rating))
  val real = data.map(item => ((item.user, item.product), item.rating))
  sqrt(
    real.join(predict).map {
      case ((user, product), (real, pre)) =>
```

```
        val err = real - pre
        err * err
      }.mean()
    )
  }
```

调用 GridSearchCV 并训练 ALS 模型，获取最终的推荐结果：

```
val (rank, lambda) = GridSearchCV (trainData, validationData)
val model = ALS.train(trainData, rank, iterations, lambda)
val index = users.cartesian(products)
val predict = model.predict(index)
```

矩阵分解模块的另一个重要功能是计算商品的相似度索引：在矩阵分解模型计算完成后，可以获得物品向量矩阵 $V=(V_1,V_2,\cdots,V_n)$，离线计算两两物品向量之间的余弦相似度，便可以获得物品相似度索引，将其保存至 MongoDB 供近线层调用：

```
val productFeatures = model.productFeatures.mapValues(new DoubleMatrix(_))
val similarities = productFeatures.cartesian(productFeatures)
  .filter { case (productA, productB) => productA._1 != productB._1 }
  .map {
    case (productA, productB) =>
    val similarity = cosineSimilarity(productA._2, productB._2)
    (productA._1, productB._1, similarity)
  }
```

其中 cosineSimilarity() 函数会返回两个向量的余弦相似度。实际上，除了计算物品向量 V_i 和 V_j 之间的余弦相似度，也可以选择计算 $\widehat{R}_i$ 和 $\widehat{R}_j$ 的余弦相似度，这样计算出来的余弦相似度，实际上是 (2.3) 式中用 Ochiia 系数计算相似度的变种，即考虑了用户对物品评分的 Ochiia 系数。

4.6 实时推荐模块

在线层在调用实时推荐模块的服务时，需要反映出用户最近一段时间对商品的偏好，基于用户在一段时间内的偏好是相似的这一假设，实时推荐模块会读取用户最近几次对商品的评分，分别计算这些商品与候选商品的相似度，并与用户的评分对应相乘，累加起来就代表对候选商品的偏好。对候选列表的所有物品都按照上述方法进行计算，选择偏好最高的商品最为用户最新的推荐列表。

具体的，记用户最近评价过的商品集合为 I，候选商品集合为 W，对于 $w\in W$，其偏好被计算为：

$$R_w = \frac{1}{|I|}\sum_{i \in I} sim(V_i, V_w) \times R_i, \tag{4.2}$$

其中 $|I|$ 代表用户最近评价过的商品数量，V_i 和 V_w 分别代表商品 i 和 w 的特征向量，R_i 代表用户对商品 i 的评分。实际上，该算法类似于基于物品的协同过滤，不同之处在于相似度的计算方法以及候选商品集合的选取。

候选商品集合 W 的选取应该遵循效率优先的原则，因为实时推荐模块位于近线层，需要在较短的时间内完成计算。本文将候选商品集合 W 选取为与用户最近一次评价过的商品 $i = |I|$ 最相近的 k 个商品（且不含用户已经评价过的商品）。这种选取对于商品推荐系统是合理的，用户既然评价了某个商品，就代表着用户对该相似的商品有着浓厚的兴趣：如果用户评价了高分，则代表用户喜欢该类商品；如果用户评价了低分，也只能代表用户对单一商品不感兴趣，因为这可能是由于包括商品质量，物流速度，客服态度在内的多种原因导致的。

实时流处理系统从 Kafka 中获取最近的一次评分流数据（可以是任何用户），将其类型转换成 org.apache.spark.streaming.dstream.Dstream[userId, productId, score, timestamp]，从 Redis 中读取当前用户最近 12 小时的物品及评分，再从 MongoDB 中读取与当前物品最相似的 20 个候选商品，即可计算出候选商品的得分，即实时推荐列表：

```
val MAX_TIME = 43200 // 12 hours
val MAX_NUM = 20
stream.foreachRDD {
  rdd =>
    rdd.foreach {
      case (userId, productId, score, timestamp) =>
        val recent = recentProducts(MAX_TIME, userId, Conf.jedis, timestamp)
        val candidate = similarProducts(MAX_NUM, productId)
        val result = score(candidate, recent)
    }
}
```

其中，recentProducts() 函数用于返回用户最近的评价，similarProducts() 函数用于返回与传入物品最相似的物品，score() 函数用于计算候选物品的得分。recentProducts() 函数可能会导致冷启动问题，因为用户 12 小时内可能没有做出任何评分，为了避免这种情况，可以选择用户最近 20 次评分作为备选。

除了推荐程序外，实时处理模块还需要另几个组件作为辅助，包括 Redis、Flume 和 Kafka。Redis 用于存储用户最近的物品评分，Flume 和 Kafka 用于完成流式数据的获取和传递。首先采用 Python 脚本模拟打点日志，每 10 秒向日志文件中写入日志，代表了

一位用户对一件商品做出了评分，这些日志将会最终被 Flume 捕捉；另外，该脚本同时需要向 Redis 缓存中存入数据，用作后续的实时推荐处理。在 Redis 中保存的数据格式为 (key,value)=(userId:userid,List(productid:score))。模拟打点日志实现如下：

```
while True:
  now = time.strftime('%Y-%m-%d %H:%M:%S', time.localtime())
  message = f'{now}  [INFO]  [{userId}|{productId}|{score}|{timestamp}])'
  with open('/home/zyl/log/logger.log', 'a') as f:
    f.write(message)
  redis_key = f'userId:{userId}'
  redis_value = f'{productId}:{score}'
  redis_client.lpush(redis_key, redis_value)
  time.sleep(10)
```

随后启动 Flume Agent 用于捕捉日志文件，该 Flume Agent 采用 exec Source（即采用 tail -f 命令读取日志内容），memory Channnel 以及 Kafka Sink。Flume 配置文件如下：

```
agent.sources = exectail
agent.channels = memoryChannel
agent.sinks = kafkasink
agent.sources.exectail.type = exec
agent.sources.exectail.command = tail -f /home/zyl/log/logger.log
agent.sinks.kafkasink.type = org.apache.flume.sink.kafka.KafkaSink
agent.sinks.kafkasink.kafka.topic = log
agent.sinks.kafkasink.kafka.bootstrap.servers = hostname:9092
agent.channels.memoryChannel.type = memory
agent.channels.memoryChannel.capacity = 10000
```

最后，Kafka 消费者会从对应的 log topic 读取数据，并用正则表达式过滤和清洗日志数据，将其转变为 (userId,productId,score,timestamp) 的四元组并传递到 recommend topic，推荐系统利用 KafkaUtils.createDirectStream 进行消费。数据过滤实现如下：

```
consumer = KafkaConsumer('log')
producer = KafkaProducer(bootstrap_servers=f'{hostname}:9092')
for message in consumer:
  line = message.value.decode('utf-8')
  pattern = re.compile(r'^[\d\s:-]+([^[]+)\[INFO\]\s+\[(\w+)\|(\w+)'
                       r'\|(\d+)\|(\d+)\].*')
  match = pattern.match(line)
  if match:
    parts = [match.group(2), match.group(3), match.group(4), match.group(5)]
    score = "|".join(parts)
    producer.send('recommend', score.encode())
```

5　商品推荐系统联调

5.1 系统初始化

首先测试系统是否初试化成功，包括各项离线/实时的各项系统组件服务是否启动成功，端口连接是否顺畅。

配置 MongoDB 的配置文件 mongodb.conf，设置数据目录以及日志目录，设定端口号为 27017，运行方式为后台运行。随后启动 MongoDB 服务器：

```
Shell> sudo ./bin/mongod -config ./data/mongodb.conf
```

标准输出中显示子进程已经 fork 成功，等待连接即代表启动完成：

```
============= mongodb start ....... =============
about to fork child process, waiting until server is ready for connections.
forked process: 1921
child process started successfully, parent exiting
```

图 5.1　MongoDB 服务器启动

可以采用如下命令访问 MongoDB 服务器，对文档数据进行操作：

```
Shell>./bin/mongo
```

标准输出显示“connect to: mongodb://hostname:27017”以及 MongoDB 的版本号即代表启动成功，如图 5.2 所示。

```
[zyl@hadoop105 mongodb]$ bin/mongo
MongoDB shell version v3.4.3
connecting to: mongodb://127.0.0.1:27017
MongoDB server version: 3.4.3
Server has startup warnings:
2023-05-05T09:07:31.602+0800 I CONTROL  [initandlisten]
2023-05-05T09:07:31.602+0800 I CONTROL  [initandlisten] ** WARNING:
2023-05-05T09:07:31.602+0800 I CONTROL  [initandlisten] **
```

图 5.2　MongoDB 服务器连接

为了方便显示，还另采用 MongoDB Compass 进行远程连接。输出正确的主机名和端口号，即可连接成功。

配置 Redis 的配置文件 redis.pid，将绑定主机 ip 设定为 0.0.0.0，便于其它节点进行访问，配置数据库存放路径和日志文件路径。随后启动 Redis 服务器：

```
Shell>./src/redis-server /etc/redis.conf
```

标准输出中并没有显示任何内容，因此采用如下命令测试与 Redis 的连接：

```
Shell>./src/redis-cli
```

观察到主机名和端口号即代表连接成功。

配置 Spark 的配置文件 slaves 和 spark-env.sh，配置 Spark 的端口号为 7077。随后启动 Spark 集群：

```
Shell>./sbin/start-all.sh
```

随后可以在浏览器中访问 WEB 端口 8080 测试 Spark 是否启动成功：

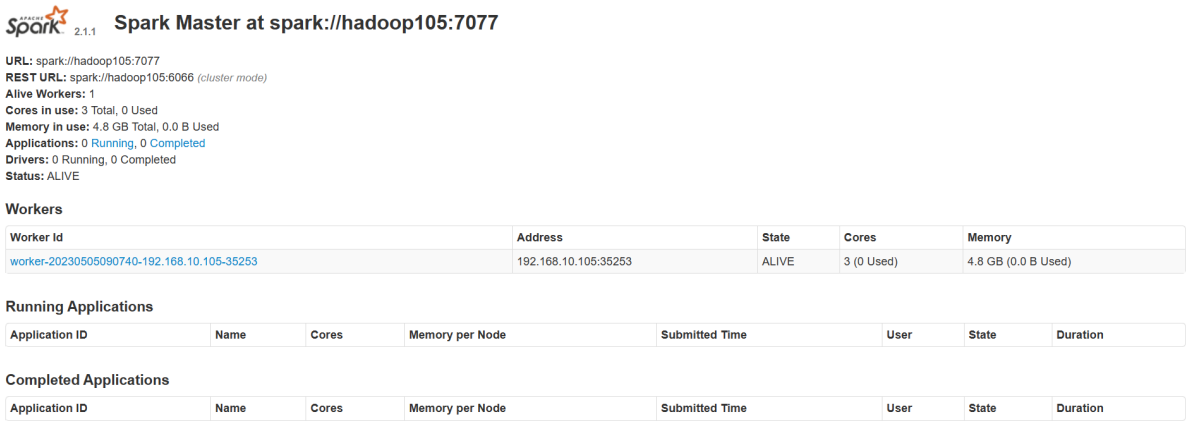

图 5.3　Spark WEB 页面

5.2 离线计算测试

运行各个离线计算模块，包括数据导入模块、统计推荐模块、内容过滤模块、协同过滤模块和矩阵分解模块，在 MongoDB Compass 中查询对应的结果。

例如，数据导入模块中写入的 product 集合，其文档内容如图 5.4 所示，代表数据被成功加载到了数据库中。

图 5.4 product 集合文档内容

例如，矩阵分解模块写入的 userRecommendations 集合，其文档内容如图 5.5 所示。图示内容表示，针对 50130 号用户可以向其推荐 139060 号商品，因为其对该商品的预期评价约为 4.7 分。不考虑_id 字段，该集合的文档内容结构为 {userId, recs:{(productId, score),…}}，这与 Spark 中 userId : RDD[userId, Seq((userId, similarity),…)] 的 RDD 结构

一致，代表推荐列表结果计算正确，在线层可以读取该集合对用户做出个性化的推荐。

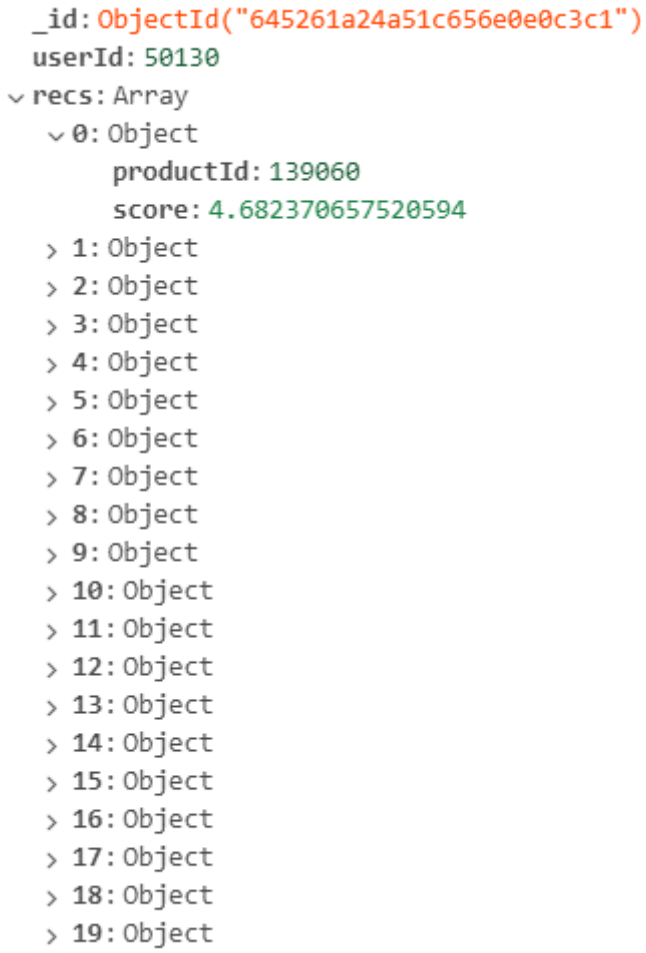

图 5.5 userRecommendations 集合文档内容

再考察物品的相似度索引，即 productSim 集合，集合文档内容如图 5.6 所示。不考虑_id 字段，该集合的文档内容结构为 {productId, recs:{(productId, score),…}}，这与 Spark 中 productId : RDD[userId, Seq((productId, similarity),…)] 的 RDD 结构依然是一致的，代表物品相似度索引计算仍然正确。

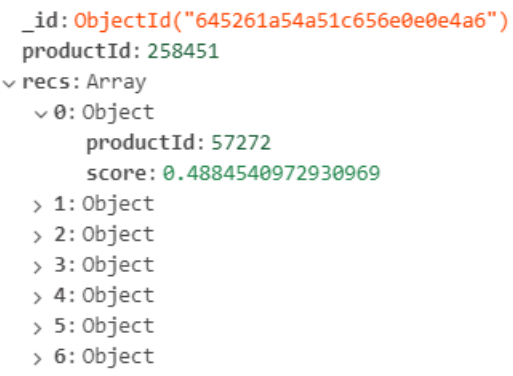

图 5.6 productSim 集合文档内容

5.3 近线计算和在线调用联调

近线计算主要是实时流处理模块，该模块设计到的组件较多，包括 MongoDB、Redis、Flume、Kafka、Spark 等，Spark 应用程序需要从 MongoDB 中读取索引数据，从 Redis 中读取用户最近几次的评分数据，从 Flume-Kafka 中读取最近一次地评分数据，最终计算推荐列表并写回 MongoDB。

首先，需要启动 Kafka 和 Flume Agent：

```
Shell>./bin/kafka-server-start.sh -daemon ./config/server.properties
Shell>./bin/flume-ng agent -c ./conf/ -f ./conf/log-kafka.properties
-n agent -Dflume.root.logger=INFO,console
```

随后，启动 Redis 客户端，并向其中写入数据，模拟用户最近的几次评分：

```
Shell>./src/redis-cli
127.0.0.1:6379 (Redis)>lpush userId:uid pid1:score1 pid2:score2 …
```

启动 Spark Streaming 程序，并利用 Kafka 创建一个生产者，通过生产者向 Spark Streaming 发送流式数据，模拟用户 uid 当前对物品 pid3 的评分。

```
Shell>./bin/kafka-console-producer.sh -broker-list hostname:9092 -
topic recommender
Kafka>uid|pid3|score3|timestamp
```

上述操作全部执行完毕后，Spark Streaming 程序就可以捕捉到该数据，并触发一次计算，为用户 uid 更新推荐列表。向 Redis 中写入的测试数据和 Spark Streaming 计算得到的数据应该相互匹配，如图 5.7 所示，即代表近线层运行成功。近线层运行成功后，在线层只需要启动 4.6 节中介绍的两个脚本程序，即可运行成功。

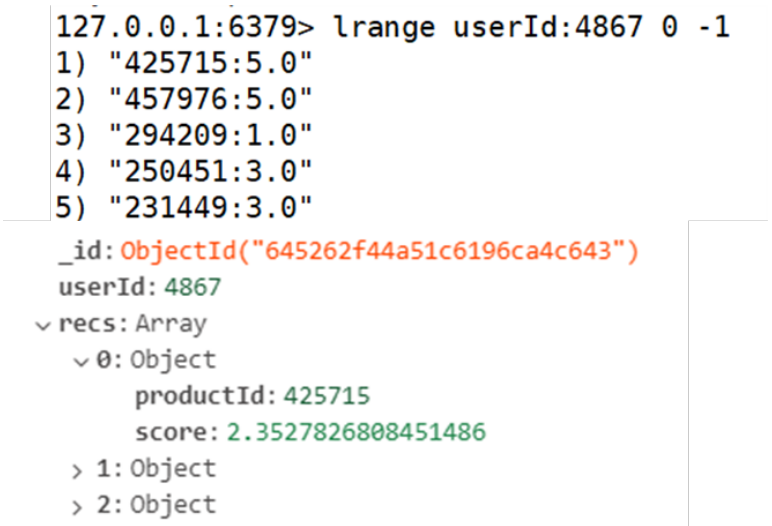

图 5.7 近线计算测试. 上部分为 Redis 数据，下部分为 Spark Streaming 程序的计算结果

6 总结与展望

随着数据信息的不断过载，用户对推荐系统的要求不断增高，推荐系统的重要性也日益增加。推荐系统，自诞生起就一直是研究的热点，本文综合了各类推荐系统算法，包括基于模型的算法（矩阵分解），基于近邻的算法（协同过滤），基于内容的算法（内容过滤），以一站式的 Spark 作为分布式计算平台，结合其他大数据处理框架，设计出了一套完整的工业化推荐系统，并对推荐算法、系统的实现和功能做出了详细地描述。本文的主要工作有：

(1) 整理工业推荐系统中最常用的推荐算法，对其进行详细地整理，例如重写 Item-CF 和 User-CF 的公式，使其更容易用函数式编程 (Scala) 的方式实现。
(2) 对于矩阵补全、矩阵分解和 ALS 算法，给出了其公式的推导过程。整理关于 Spark 矩阵运算和优化的论文，给出了 ALS 算法在分布式内存计算框架中的实现方式。
(3) 对 Netflix 的推荐系统三层架构进行了剖析，将私有化的组件以及过时的组件，更换为最新的开源的组件，例如将 EVCache 更改为 Redis，将 HDFS 更改为 MongoDB，将私有组件更改为 Flume、Kafka 等。
(4) 采用 Scala 语言，给出了推荐系统常用算法的函数式编程实践，相比于传统的面向对象或面向过程的编程方式，Scala 语言写出的推荐算法效率更高，更适合于在 Spark 环境下运行。

当然，推荐系统领域已经持续发展了约 30 年，并且是一个工业界水平显著高于学术界的领域，大数据领域也存在同样的问题，因此本文实现的推荐系统虽然尽可能做到了结构的完整，但是相比工业界的推荐系统（例如淘宝、小红书、豆瓣等），其推荐的质量和效率、安全都有着较大的差距，有待进一步的改进。以下是一些可能的改进方向：

(1) 在工业界，常常结合更多的算法，实现数据流的混合，即实现推荐系统的 Pipeline 架构：召回-粗排-精排-重排，本文涉及到的算法，只可能覆盖召回、粗排和精排，而不涉及重排，因此可以考虑加入与用户体验严重相关的重排算法，提高推荐的质量。
(2) 增强推荐系统的可解释性。对于用户来说，他们更关注推荐系统生成的推荐结果背后的逻辑和原因[19]。因此推荐系统需要具备一定的可解释性，以保证用户对推荐结果的理解和信任度[20]，同时也为推荐系统的优化和改进提供了指导方向。
(3) 推荐系统的安全性也是需要重视的问题。随着推荐系统在金融、医疗、保险等领域的应用，推荐系统的安全性也成为了一个关键问题[21]。例如，如何保护用户隐私，如何防止恶意攻击等都需要在推荐系统设计和实现中予以考虑。在推荐系统中加入联邦学习和密码学是一个可行的方向[22]。

参考文献

[1] Cacheda F, Carneiro V, D Fernández, et al. Comparison of collaborative filtering algorithms: Limitations of current techniques and proposals for scalable, high-performance recommender systems[J]. ACM Transactions on the Web, 2011, 5(1):1-33.

[2] Zhu S A, Qg A, Jie Y B, et al. Research commentary on recommendations with side information: A survey and research directions:, 10.1016/j.elerap.2019.100879.

[3] Gower S. Netflix prize and SVD[D]. University of Puget Sound, 2014.

[4] Adomavicius G, Tuzhilin A. Toward the next generation of recommender systems: a survey of the state-of-the-art and possible extensions[J]. IEEE Transactions on Knowledge and Data Engineering, 2005, 17(6):734-749.

[5] Sarwar B, Karypis G, Konstan J, et al. Item-based collaborative filtering recommendation algorithms[C]//Proceedings of the 10th international conference on World Wide Web. 2001: 285-295.

[6] Su X, Khoshgoftaar T M. A Survey of Collaborative Filtering Techniques[J]. Advances in Artificial Intelligence, 2009, 2009(12): 1-19.

[7] Yue S, Larson M, Hanjalic A. Collaborative Filtering beyond the User-Item Matrix: A Survey of the State of the Art and Future Challenges[J]. ACM Computing Surveys (CSUR), 2014, 47(1):1-45.

[8] 陈蕾,陈松灿.矩阵补全模型及其算法研究综述[J].软件学报,2017,28(06):1547-1564.DOI:10.13328/j.cnki.jos.005260.

[9] Hastie T, Mazumder R, Lee J, et al. Matrix Completion and Low-Rank SVD via Fast Alternating Least Squares[J]. Journal of Machine Learning Research, 2014, 16(1):3367-3402.

[10] Zaharia M, Chowdhury M, Das T, et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing[C]// Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation. USENIX Association, 2012.

[11] Bosagh Zadeh R, Meng X, Ulanov A, et al. Matrix computations and optimization in apache spark[C]//Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. 2016: 31-38.

[12] 覃雄派,王会举,杜小勇,王珊.大数据分析——RDBMS 与 MapReduce 的竞争与共生[J].软件学报,2012,23(01):32-45.

[13] M Armbrust, RS Xin, Cheng L, et al. Spark SQL: Relational Data Processing in Spark[C]// International Conference on Management of Data. ACM, 2015.

[14] Carbone P, Katsifodimos A, Ewen S, et al. Apache Flink: Stream and Batch Processing in a Single Engine[J]. IEEE Data Eng. Bull., 2015, 38:28-38.

[15] 程学旗,靳小龙,王元卓,郭嘉丰,张铁赢,李国杰.大数据系统和分析技术综述[J].软件学报,2014,25(09):1889-1908.DOI:10.13328/j.cnki.jos.004674.

[16] 孙大为,张广艳,郑纬民.大数据流式计算:关键技术及系统实例[J].软件学报,2014,25(04):839-862.DOI:10.13328/j.cnki.jos.004558.

[17] 申德荣,于戈,王习特,聂铁铮,寇月.支持大数据管理的 NoSQL 系统研究综述[J].软件学报,2013,24(08):1786-1803.

[18] Zaharia M, Das T, Li H, et al. Discretized Streams: Fault-Tolerant Streaming Computation at Scale[C]// Twenty-fourth Acm Symposium on Operating Systems Principles. ACM, 2013.

[19] 魏楚元,王梦珂,户传豪,张桃齐.增强推荐系统可解释性的深度评论注意力神经网络模型[J/OL].计算机应用:1-8[2023-05-05].http://kns.cnki.net/kcms/detail/51.1307.TP.20230423.1140.002.html

[20] 熊剑. 基于 Spark 的可解释性推荐系统研究与应用[D].沈阳工业大学,2021.DOI:10.27322/d.cnki.gsgyu.2021.000871.

[21] Chai D , Wang L , Chen K , et al. Secure Federated Matrix Factorization[J]. Intelligent Systems, IEEE, 2020, PP(99):1-1.

[22] Lin G , Liang F , Pan W , et al. FedRec: Federated Recommendation with Explicit Feedback[J]. Intelligent Systems, IEEE, 2020, PP(99):1-1.