第 7 章 重排模块

在上一章中，我们详细介绍了级联式推荐系统中的精排模块。精排阶段通过更复杂的用户行为建模与多目标优化方法，对粗排筛选后的候选集合进行精细化打分，从而在单个候选物品层面实现较为准确的价值评估。然而，从系统整体视角来看，精排的输出本质上仍然是一个**基于独立打分的排序结果**，其优化目标主要集中在“单个物品是否优质”，而非“整个推荐序列是否最优”。

当推荐系统进入信息流、多内容连续消费等应用场景时，用户的真实体验往往由**一组内容的组合效果**共同决定，而不仅仅取决于单个内容的质量。这种从“单点决策”到“序列决策”的建模差异，使得仅依赖精排阶段的排序策略，在工业实践中逐渐暴露出其局限性。因此，在精排之后引入重排（re-ranking）模块，对排序结果进行进一步的序列级优化，成为工业级推荐系统中的重要设计选择。

基于上述背景，本章将重点介绍重排模块的设计动机、核心处理流程以及典型实现范式，并分析其在不同业务场景中的必要性与适用边界。

7.1 为什么需要重排

在工业级推荐系统中，重排（re-ranking）模块通常位于精排（full-ranking）模块之后，其核心作用是在精排输出的候选列表基础上，引入业务规则与策略目标进行二次调整。典型的应用包括提升结果多样性、控制广告曝光优先级、调节流量分发等，以确保最终推荐结果更契合平台的整体运营目标。然而，读者可能会产生一个根本性的疑问：**为何需要专门设置重排模块?** 要回答这一问题，关键在于厘清精排与重排在目标与方法上的本质差异。

传统精排模块通常采用复杂的深度神经网络（Deep Neural Network, DNN）对每个候选物品进行**单点式**（point-wise）打分，并依据得分高低直接排序。这种策略本质上是一种**贪心选择**——它仅追求每个位置上的局部最优，而忽视了整个推荐序列的全局效果。以短视频推荐场景为例（如图 7.1 所示），用户在应用中通过上下滑动浏览内容，系统依次展示短视频。出于排版示意，图中将视频横向排列，但实际交互是纵向连续的。在此产品形态下，用户通常无法预知下一个视频的内容，其在单次会话（session）中消费的视频数量，与视频序列的整体排列顺序密切相关。若排序不佳，用户可能在浏览几条后便退出，导致人均消费深度下降，损害用户体验。

图 7.1: 短视频序列示意图。

此外，观察数据还揭示了一个显著现象：随着用户在会话中浏览位置的后移，其单条视频的观看时长往往呈递减趋势。这反映出强烈的**位置偏置**（position bias）：即用户对靠前位置的内容天然赋予更高注意力，而对后续内容的耐心逐渐衰减。这种由产品交互形态引发的用户行为偏差，是推荐系统在分发策略中必须重点考量的因素。正是由于上下滑动的产品设计及其带来的位置偏置特性，工业界实践中发现：仅依赖精排阶段的单点打

分排序，难以实现全局最优的推荐效果，极易陷入局部最优陷阱。因此，重排模块应运而生，其核心目标是从**整个推荐序列的视角**出发，进行更全局、更协同的排序优化。

值得进一步追问的是：**为何不在精排阶段就直接进行序列级的全局优化？**答案在于计算复杂度与工程可行性的权衡。假设精排阶段的候选集规模为 500，而最终需展示的视频数量为 8，则可能的排列组合总数高达 500^8，搜索空间极其庞大，难以在毫秒级响应要求下完成优化。相比之下，在精排已筛选出 top-100 的高质量候选后，仅从仅对这 100 个物品的顺序进行重排，其优化空间显著缩小（100^8）。为进一步压缩搜索空间，也可以将候选集截断至最终的 top K ($K < 10$) 之后再进行重排序，此时最多仅有 $K!$ 种排列。这一策略既大幅降低了计算开销，又能在有限延迟内实现更优的序列级决策。因此，重排模块的引入，是在保证系统效率的前提下，对推荐结果进行精细化、策略化与全局化调优的关键一环。

7.2 重排模块架构

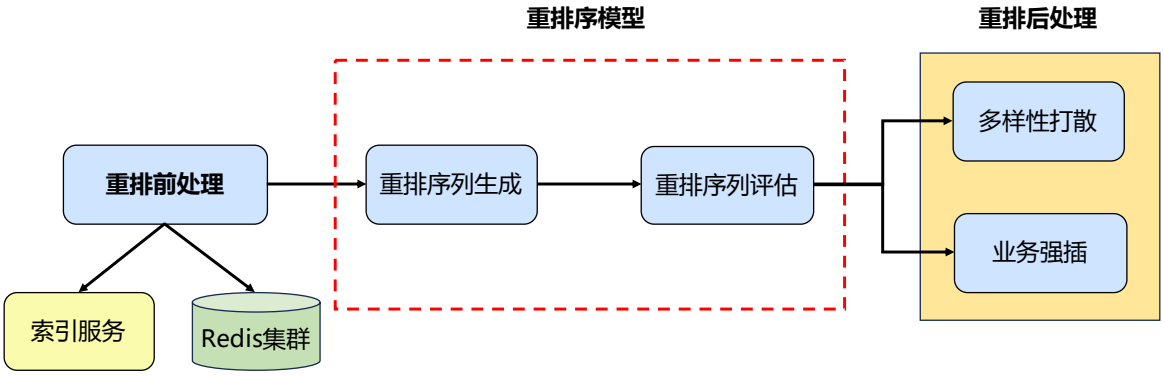

图 7.2: 重排模块内部处理流程。

重排模块（Reranking）的整体流程如图 7.2 所示，通常由四个关键环节构成：**重排前处理（Pre-processing）**、**序列生成（List Generation）**、**序列评估（List Evaluation）**以及**多样性打散（Diversification）**。其中，重排前处理阶段主要负责从外部正排索引服务、Redis 集群以及各类在线特征存储系统中获取重排所需的特征信息，并完成候选物品的属性初始化、特征补全以及必要的数据清洗与预处理工作，从而为后续序列级建模提供统一且高质量的输入。

在整体框架中，序列生成与序列评估构成了重排模块的核心计算主干。目前工业界主流方案通常采用经典的**两阶段重排架构（Two-Stage Reranking Framework）**，即“序列生成（List Generator）+ 序列评估（List Evaluator）”的范式。在该框架下，序列生成阶段负责在候选集合上构造若干可能的排序序列，而序列评估阶段则对这些候选序列进行整体打分与选择，从而在用户体验、业务目标与内容生态约束之间实现综合权衡。

与召回、粗排及精排阶段主要针对单个物品（item-level）价值评估不同，重排阶段的优化对象已上升至**序列级（list/slate-level）决策问题**。因此，重排不仅需要考虑单个物品的质量，还必须显式建模物品之间的相互关系。例如，相邻内容的相似性控制、创作者分布的均衡性、内容垂类的覆盖度，以及广告、直播、电商等不同类型内容在序列中的插入策略等，均属于重排阶段需要重点处理的问题。

近年来，随着生成式模型与大语言模型的发展，学术界与工业界也开始探索新的重排范式。例如，一类方法采用**一阶段生成式重排（One-Stage Reranking）**，直接建模并生成最终推荐序列；另一类方法则基于**生成式重排网络（Generative Reranking Network, GRN）**，通过序列生成模型显式刻画复杂的依赖关系与全局结构。这类方法在表达能力上更强，但由于其对计算资源与系统稳定性的要求较高，目前在大规模工业系统中仍未完全替代传统的两阶段架构。相关前沿方法将在后续第 13章与第 34章中进一步展开讨论。

此外，需要指出的是，重排模块并非所有推荐系统的标准组成部分，其是否引入取决于具体业务形态与优化目标。当推荐系统面向短视频、信息流或图文流等连续消费场景时，用户体验由多个内容的组合共同决定，此时系统需要在序列层面同时优化多样性、结构性与长期收益，重排模块通常是必要组件。然而，在直播推荐、广

告推荐或电商活动等场景中，推荐结果往往仅作为主信息流中的少量补充内容存在，此时系统更多关注单个候选的匹配质量，而较少涉及序列级优化，因此通常可以弱化或省略独立的重排模块设计。重排模块的存在与否，并不取决于系统架构的形式，而取决于业务目标是否从**单物品最优（item-optimal）**进一步扩展至**序列整体最优（list-optimal）**。当优化目标从单点点击率提升扩展到用户整体浏览体验与内容结构优化时，重排模块的价值便会显著体现。

从数学的建模的角度来看，重排模块可以被抽象为一个序列优化问题，如下所示：

定义 7.1

给定大小为 N 的候选集合 $\mathcal{V}$，重排的目标是在满足长度约束 T 的条件下，寻找最优序列 $\pi = \{i_1, i_2, ..., i_T\}$，使得整体目标函数最大化：

$$\max_{\pi} \mathcal{F}(\pi) = \sum_{t=1}^{T} \text{Rel}(i_t, u) + \lambda \cdot \text{Div}(\pi) + \mu \cdot \text{Constrain}(\pi) \tag{7.1}$$

其中，$\text{Rel}(\cdot)$ 表示相关性函数，用于衡量物品与用户兴趣的匹配程度；$\text{Div}(\cdot)$ 表示多样性函数，用于约束序列内部冗余性；$\text{Constrain}(\cdot)$ 表示业务约束项，例如广告配比、作者分布或内容策略等。

从优化形式上看，该问题本质上是在一个指数级的排列空间中进行搜索。给定候选集合大小为 N，当 $T \ll N$ 时，其可行解空间规模为：

$$\mathcal{O}(P(N,T)) = \mathcal{O}\left(\frac{N!}{(N-T)!}\right), \tag{7.2}$$

当 T 进一步较小（例如工业系统中常见的 $T \leq 10$）时，也可近似认为搜索空间规模在 N^T 级别，整体呈现出明显的组合爆炸特性。

因此，从本质上来看，重排问题可以视为一个**受约束的序列组合优化问题**，其目标是在难以穷举的搜索空间中寻找近似最优解。由于该问题同时涉及离散选择与非线性目标函数优化，在一般情况下难以通过精确算法在在线环境中求解，因此工业界通常采用贪心搜索、束搜索、子模 (submodular) 优化或启发式生成等方式对其进行近似求解，从而在计算效率与排序效果之间取得平衡。

7.3 序列生成

在序列生成过程中，本质上是在序列的高维空间中搜索潜在的最优序列集合。由于搜索空间依然较大，通常难以进行暴力枚举，线上系统的算力与延迟约束无法承受。因此，序列生成常被视为**序列维度的召回阶段**。目前工业界常用的序列生成方法众多，其中两类经典策略被广泛采用。一类是**基于多目标融合排序的贪心序列生成方法**，另一类是**基于束搜索（beam search）的序列生成方法**。基于多目标融合排序的贪心序列生成方法通过对每个候选物品在多个业务目标（如有效播放率 pevr、长播率 plvr、点赞率 pltr、评论率 pcmr、转发率 pftr 等）上的预估值进行加权融合，得到一个综合得分，并依此直接排序生成推荐序列。以短视频场景为例，假设候选集包含 5 个物品，其各项指标的预估值如表7.1 所示。

表 7.1: 重排序多目标融合序列生成阶段物品示例 pxtr 预估值。

物品 ID	预估有效播放率 pevr	预估长播率 plvr	预估点赞率 pltr	预估评论率 pcmr	预估转发率 pftr
item_0	0.374540	0.020584	0.611853	0.607545	0.122038
item_1	0.950714	0.969910	0.139494	0.170524	0.495177
item_2	0.731994	0.832443	0.292145	0.065052	0.034389
item_3	0.598658	0.212339	0.366362	0.948886	0.909320
item_4	0.156019	0.181825	0.456070	0.965632	0.258780

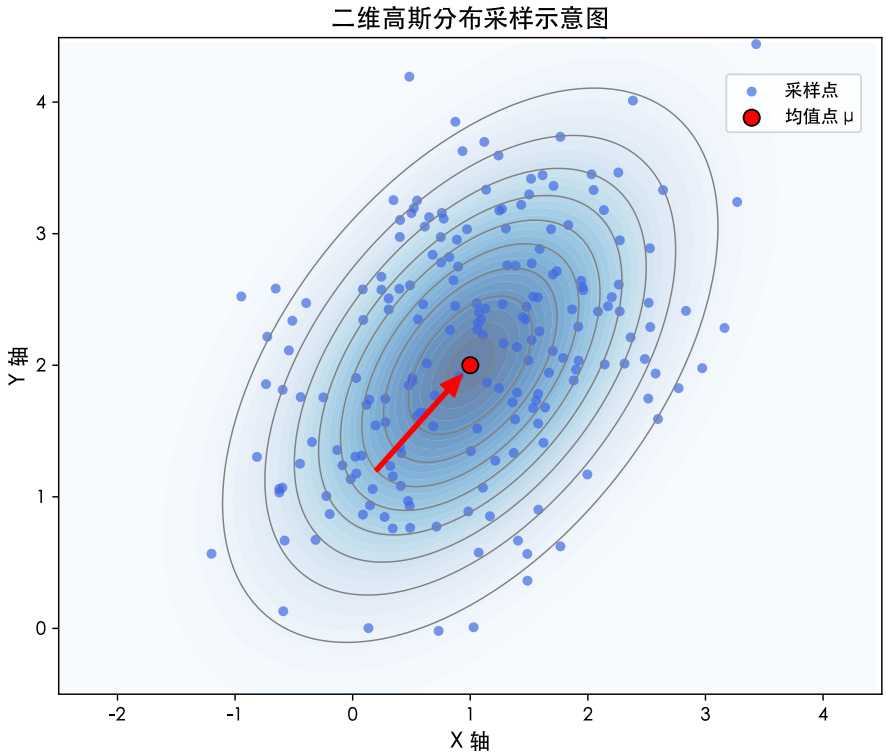

图 7.3: 多维高斯分布采样示意图。

对于上述的候选物品，可以设置一个排序的融合公式，比如线性融合的排序公式：

$$\text{final\_score} = w_1 \times \text{p\_evr} + w_2 \times \text{p\_lvr} + w_3 \times \text{p\_ltr} + w_4 \times \text{p\_cmr} + w_5 \times \text{p\_ftr} \tag{7.3}$$

然后根据 final_score 即可进行排序得到一个贪心策略生成的序列。由于在上述排序公式中有 w_1、w_2、w_3、w_4、w_5 这 5 个参数，因此，我们可以将其整合成一个 5 维的参数向量，即

$$\mathbf{w} = [w_1, w_2, w_3, w_4, w_5] \tag{7.4}$$

在实际应用中，可首先根据人工经验设定权重向量 $\mathbf{w}$ 的基准值，并将其作为均值向量，通过多元高斯分布进行采样，从而生成多组不同的权重组合，以获得多样化的排序结果。为便于读者直观理解高斯采样的含义，图 7.3 以二维高斯分布为例，展示了在参数向量附近进行扰动采样的过程。图中红色点表示初始权重向量的位置，即高斯分布的均值向量；周围的蓝色点则为从该多元高斯分布中采样得到的样本点。可以观察到，距离红色点越远的采样点数量越少，体现了高斯分布在均值附近具有更高概率密度的特性。

详细的算法代码实现如下7.1和7.2所示。在这里首先我们可以随机初始化物品的 pxtr 特征，注意在实际应用中需要替换成真实的 pxtr。

代码 7.1: 基于多目标融合排序的贪心序列生成算法代码（初始化）

```
import numpy as np
import pandas as pd

# ============ 模拟物品特征 ============
num_items = 10
np.random.seed(42)

# 每个物品的5个特征: p_evr, p_lvr, p_ltr, p_cmr, p_ftr
```

```
items = pd.DataFrame({
    'item_id': [f'item_{i}' for i in range(num_items)],
    'p_evr': np.random.rand(num_items),
    'p_lvr': np.random.rand(num_items),
    'p_ltr': np.random.rand(num_items),
    'p_cmr': np.random.rand(num_items),
    'p_ftr': np.random.rand(num_items)
})
print("示例物品特征: ")
print(items.head())
```

接下来我们可以定义高斯采样的数据分布并基于高斯采样之后的权重来生成重排序序列。其中，协方差矩阵作为超参数控制探索范围：协方差越小，采样越集中；反之则探索更广。

代码 7.2: 基于多目标融合排序的贪心序列生成算法代码（采样与序列生成）

```
# ============ 定义高斯采样 ============
mean = np.array([0.2, 0.3, 0.2, 0.2, 0.1]) # 权重均值
cov = np.diag([0.01]*5)                # 协方差矩阵（独立采样）
num_samples = 5                         # 生成5组不同的权重组合
weights_samples = np.random.multivariate_normal(mean, cov, num_samples)
print("\n采样得到的权重向量: ")
print(weights_samples)

# ============ 基于权重生成重排序序列 ============
rerank_sequences = []
for i, w in enumerate(weights_samples):
    # 计算每个物品的 final_score
    items['final_score'] = (
        w[0]*items['p_evr'] + w[1]*items['p_lvr'] + w[2]*items['p_ltr'] + w[3]*items['p_cmr'] + w[4]*items['p_ftr']
    )
    # 根据 score 排序
    ranked = items.sort_values(by='final_score', ascending=False)['item_id'].tolist()
    rerank_sequences.append({
        'sample_id': i, 'weights': w, 'ranking': ranked
    })

# ============ 输出示例 ============
for seq in rerank_sequences:
    print(f"\n采样{seq['sample_id']}权重:{np.round(seq['weights'],3)}")
    print(f"排序结果:{seq['ranking']}")
```

尽管该方法利用了多目标信息，但其本质仍是贪心策略，每一步仅选取当前得分最高的物品。虽然计算高效，却易陷入局部最优。因此，需依赖大量采样以扩大覆盖范围。为在贪心搜索与穷举搜索之间取得平衡，**束搜索**（beam search）成为一种更具探索能力的替代方案。

束搜索算法最早由 Lowerre 在 1976 年的语音识别系统 Harpy 中提出 [9]，随后 Rubin 与 Reddy 在 1977 年 IJCAI 会议论文中将其形式化为一种非回溯的确定性启发式搜索方法 [12]。此后，Beam Search 被广泛应用于机器翻译 [3, 13]、文本生成 [11]、强化学习序列建模 [6] 等多个领域。

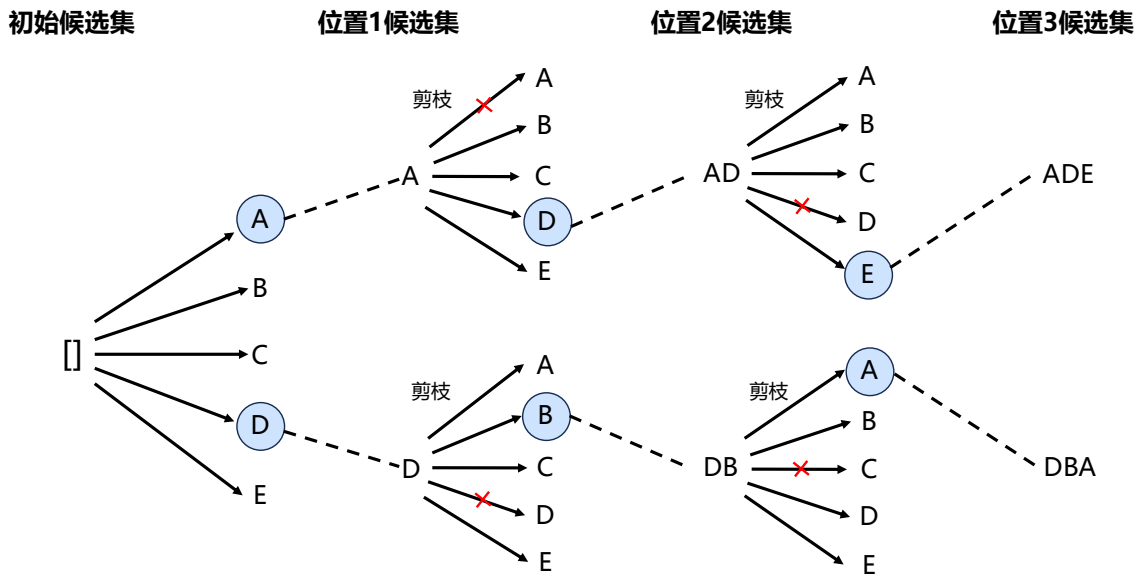

图 7.4: Beam Search 算法基本原理示意图。

如图7.4 所示，Beam Search 在每一步扩展时保留 top-K 条候选路径（即“束宽”），并在搜索过程中对不合法序列进行剪枝。例如，可嵌入业务规则：相同垂类内容不得连续出现。此类约束可在中间步骤提前剔除无效路径，提升搜索效率。下面的代码展示了一种融合高斯采样与 Beam Search 的序列生成实现：在每一步不仅扩展候选序列，还动态采样新的权重向量，以增强探索能力。当然，也可在搜索全程固定权重，但会牺牲部分多样性。

首先仍然是每个物品 pxtr 的初始化和相关 Python 库函数的导入，如7.3所示。

代码 7.3: 基于 Beam Search 的序列生成算法代码（初始化）

```
import numpy as np
import pandas as pd

# ============ 模拟物品特征 ============
num_items = 10
np.random.seed(42)

# 每个物品的5个特征: p_evr, p_lvr, p_ltr, p_cmr, p_ftr
items = pd.DataFrame({
    'item_id': [f'item_{i}' for i in range(num_items)],
    'p_evr': np.random.rand(num_items),
    'p_lvr': np.random.rand(num_items),
    'p_ltr': np.random.rand(num_items),
    'p_cmr': np.random.rand(num_items),
    'p_ftr': np.random.rand(num_items)
})

print("示例物品特征: ")
print(items.head())
```

接下来，我们定义权重向量，并且在 beam search 的过程中不断更新权重向量，同时完成序列的生成。以下7.4是 beam search 和序列生成部分的核心代码。

代码 7.4: 基于 Beam Search 的序列生成算法代码（beam search）

```
# ============ 定义权重参数 ============
mean = np.array([0.2, 0.3, 0.2, 0.2, 0.1]) # 权重均值
cov = np.diag([0.01]*5)                # 协方差矩阵（独立采样）

# ============ 基于Beam Search生成重排序序列 ============
def beam_search_rerank(items, mean, cov, beam_size=3, num_steps=5):
    # 初始化beam: 每个元素为(当前序列, 当前分数, 权重)
    initial_weights = np.random.multivariate_normal(mean, cov, beam_size)
    beam = []

    for w in initial_weights:
        # 计算初始分数
        items['final_score'] = (
            w[0]*items['p_evr'] + w[1]*items['p_lvr'] + w[2]*items['p_ltr'] + w[3]*items['p_cmr'] + ↩
                ↪ w[4]*items['p_ftr']
        )
        # 初始排序
        ranked = items.sort_values(by='final_score', ascending=False)['item_id'].tolist()
        # 计算序列总分数（使用topN分数和）
        total_score = items['final_score'].nlargest(len(ranked)).sum()
        beam.append((ranked, total_score, w))
    #  beam search迭代优化
    for step in range(num_steps):
        candidates = []
        # 扩展当前beam中的每个候选

        for seq, score, w in beam:
            # 生成新的权重候选

            new_weights = np.random.multivariate_normal(w, cov, beam_size)
            # 基于新的权重进行序列生成
            for new_w in new_weights:
                # 基于新权重计算分数
                items['final_score'] = (
                    new_w[0]*items['p_evr'] + new_w[1]*items['p_lvr'] + new_w[2]*items['p_ltr'] + ↩
                        ↪ new_w[3]*items['p_cmr'] + new_w[4]*items['p_ftr']
                )
                # 生成新排序
                new_ranked = items.sort_values(by='final_score', ascending=False)['item_id'].tolist↩
                    ↪ ()
                new_score = items['final_score'].nlargest(len(new_ranked)).sum()
                candidates.append((new_ranked, new_score, new_w))

        # 筛选出最优的beam_size个候选
        candidates.sort(key=lambda x: x[1], reverse=True)
        beam = candidates[:beam_size]
    return beam
```

最后是整个基于 beam search 的序列生成算法代码的程序入口，包括了对于整个 beam search 函数的调用以及输出最终生成的序列结果，如下7.5所示。

代码 7.5: 基于 Beam Search 的序列生成算法代码（程序入口）

```
# 运行beam search
beam_size = 3
num_steps = 5
best_sequences = beam_search_rerank(items, mean, cov, beam_size, num_steps)
# ============ 输出结果 ============
print(f"\nBeamSearch结果(beam_size={beam_size},steps={num_steps})")
for i, (ranking, score, weights) in enumerate(best_sequences):
    print(f"\n候选{i+1}权重:{np.round(weights,3)}")
    print(f"序列分数:{np.round(score,3)}")
    print(f"排序结果:{ranking}")
```

7.4 序列评估与排序

重排序的序列生成过程通常由多种方法并行提供候选序列，而序列评估则相对固定，普遍采用监督学习范式对整个序列的价值进行建模。以短视频上下滑场景为例，最简化的评估方式是将序列中每个物品的单点分数 $f(x_i)$ 按位置衰减加权求和。这里的分数 $f(x_i)$ 可以是单一的标签，比如观看时长，也可以是多个目标经过线性或非线性融合之后的融合分数。考虑到位置偏置，权重可设为 $w_i = \frac{1}{i}$ 或指数衰减形式 $w_i = e^{-(i-1)}$，则序列价值定义为：

$$\text{value} = \sum_{i=1}^{N} w_i \times f(x_i) \tag{7.5}$$

基于此，可采用回归损失进行训练：

$$\mathcal{L} = \frac{1}{B} \sum_{i=1}^{B} \|\hat{y}_i - \text{value}_i\|^2 \tag{7.6}$$

其中 B 是整个序列数据的批次大小。

更简化的方式是直接以序列整体观看时长 y 为标签，并通过阈值转化为二分类任务：

$$\mathcal{L} = -\frac{1}{B} \sum_{i=1}^{B} y_i \log \hat{y}_i + (1 - y_i) \log (1 - \hat{y}_i)) \tag{7.7}$$

第三种更贴合产品逻辑的评估方式，基于用户“是否继续下滑”的行为建模曝光概率。设用户打开 App 的初始概率为 p_0，观看第 j 个视频后继续下滑的概率为 $p_{\text{slide}}(x_j)$，则第 i 个物品的曝光概率为：

$$p(\text{item}_i) = \begin{cases} p_0, & i = 1 \\ p_0 \prod_{j=1}^{i-1} p_{slide}(x_j), & i > 1 \end{cases} \tag{7.8}$$

相应地，第 i 个物品的价值为：

$$\text{value}_i = p(\text{item}_i) \times f(x_i) \tag{7.9}$$

而整个序列 L 的整体价值定义为：

$$\text{value} = \sum_{i=1}^{N} \text{value}_i \tag{7.10}$$

在实际训练中，若序列价值为各物品价值之和，则模型重点在于准确预估每个位置的物品价值。此时可对每个物品分别施加损失：

$$\begin{aligned} \mathcal{L} &= \frac{1}{B} \sum_{i=1}^{B} \sum_{j=1}^{N} \mathcal{L}_{ij} \\ \mathcal{L}_{ij} &= \|\hat{y}_{ij} - \text{value}_{ij}\|^2 \end{aligned} \tag{7.11}$$

从强化学习的视角来看，重排模块中的序列生成（List Generator）可以类比为 **Actor**，负责在巨大的序列空间中生成候选排序方案；而序列评估（List Evaluator）则对应 **Critic**，用于评估不同排序序列的质量，并为生成策略提供优化信号。近年来，强化学习在序列生成阶段得到了广泛应用，例如利用策略梯度（Policy Gradient）、蒙特卡洛树搜索（Monte Carlo Tree Search, MCTS）等方法提升序列搜索效率和生成质量。

然而，完全使用强化学习替代传统重排流程在工业界仍面临诸多挑战，例如 Q 值估计不稳定、样本效率较低、训练过程难以收敛，以及线上探索带来的业务风险等问题。因此，目前大多数工业界推荐系统仍然采用“序列生成 + 序列评估”的两阶段重排架构，而强化学习更多作为其中某些关键环节的优化手段进行局部应用。关于强化学习在推荐系统中的具体实践，包括强化学习重排、流量调控、排序公式超参数寻优以及广告竞价等典型应用，我们将在第 28章中进行详细介绍。

7.5 多样性打散

重排模块的最后一个阶段是**重排后处理**（post-processing），其核心任务是在序列评估选出的最优候选序列基础上，进一步嵌入强约束性业务逻辑，以确保最终输出的推荐结果既符合算法优化目标，又满足产品运营与用户体验的多维要求。其中，最为关键的两个环节是**多样性打散**（diversity diversification）与**业务强插**（business-level forced insertion）。

多样性打散旨在缓解推荐结果中内容同质化的问题，避免系统在短时间内向用户密集推送同类内容，从而引发“信息茧房”效应或审美疲劳。大量实践表明，若相同类别（如影视、美食、萌宠等）的内容在推荐列表中连续或高频出现，用户极易产生倦怠感，进而降低滑动意愿、缩短单次使用时长，甚至影响长期留存。因此，引入多样性机制不仅是提升用户体验的重要手段，更是保障平台核心指标（如人均观看时长、次日留存率等）的关键策略。

目前，工业界主流的多样性打散方法可分为两大类：**基于规则的多样性打散**与**基于模型的多样性打散**。**基于规则的多样性打散**逻辑清晰、实现简单、线上可控性强，是大多数推荐系统初期或对稳定性要求较高的场景中的首选方案。其典型做法是对特定内容类别在推荐列表中的出现频次或位置间隔施加硬性约束。例如，在单次推荐列表长度为 10 的短视频场景中，可设定“10 出 1”规则：即最多允许 1 个短剧类长视频出现在该列表中。更进一步，还可引入**跨刷次的滑动窗口约束**，例如在用户最近滑动的 20 个视频构成的窗口内，影视类内容不得超过 3 个。此类规则虽显“粗粒度”，但因其可解释性强、调试便捷，常被用于兜底保障，尤其适用于广告、热点、合规等强业务诉求场景。

相比之下，**基于模型的多样性打散**则试图在保持相关性的同时，以更精细、更自适应的方式建模内容间的差异性。这类方法通常将多样性视为一个可优化的目标函数，与相关性进行联合权衡。其中，三种经典算法被广泛研究与应用：

- **MMR**（Maximal Marginal Relevance，最大边缘相关）[1]：通过在每一步选择与用户兴趣最相关、同时与已选内容最不相似的物品，实现相关性与多样性的贪心平衡。其核心思想是最大化边际收益，形式简洁，易于嵌入现有排序流程。
- **DPP**（Determinantal Point Process，行列式点过程）[7]：一种基于概率点过程的子集选择模型，能够自然地

刻画物品间的“排斥性”（repulsion），从而在全局范围内生成高相关性且高多样性的推荐子集。尽管 DPP 理论优美，但其计算复杂度较高，通常需结合采样或近似算法以适配线上延迟要求。

- **SSD**（Sliding Spectrum Decomposition，滑动频谱分解）[5]：一种面向序列推荐场景的高效多样性算法，通过在滑动窗口内对内容特征进行谱分解，动态识别并抑制冗余模式。SSD 特别适合处理长序列、高吞吐的推荐场景，在保证多样性的同时对系统性能影响较小。

在实际系统中，上述方法往往并非孤立使用，而是根据业务阶段、资源预算与效果目标进行组合。例如，可先通过规则打散进行粗筛，再以 MMR 或 SSD 进行细粒度优化；或在 DPP 生成候选子集后，叠加业务强插逻辑（如插入运营活动视频、公益内容等），确保策略落地。这种“规则兜底 + 模型优化 + 业务干预”的多层次后处理架构，已成为现代工业级推荐系统重排阶段的标准范式。

下面我们重点介绍一下 MMR、DPP 和 SSD 这三种多样性打散算法以及它们之间的区别。

7.5.1 MMR 算法

MMR（Maximal Marginal Relevance，最大边际相关性）算法最早由 Carbonell 与 Goldstein 于 1998 年在论文《The Use of MMR, Diversity-Based Reranking for Reordering Documents and Producing Summaries》中提出 [1]。其核心思想是在文本检索重排序与摘要生成任务中，同时兼顾**查询相关性**与**信息新颖性**，从而在保留关键信息的同时有效减少结果冗余，尤其在多文档摘要任务中展现出显著优势。从向量空间的角度理解，所谓“信息新颖性”即体现为候选内容与已选内容之间的**低相似性**（或高不相似性）。

设 C 为物品的候选集合，Q 为查询，经初始信息检索后得到的结果集记为 $R = IR(Q, C, \theta)$。令 S 表示当前已从 R 中选出的子集，则 $R \setminus S$ 表示尚未被选中的剩余候选。在此设定下，MMR 的优化目标可形式化为：

$$MMR \overset{def}{=} \arg \max_{D_i \in R \setminus S} \left[\lambda \left(Sim_1 \left(D_i, Q \right) - (1 - \lambda) max_{D_j \in S} Sim_2 \left(D_i, D_j \right) \right) \right] \tag{7.12}$$

其中，$\lambda \in [0, 1]$ 是一个超参数，用于平衡**边际相关性**（Marginal Relevance）与**相关性创新度**（Relevant Novelty）；Sim_1 衡量候选文档 D_i 与查询 Q 的相关性，Sim_2 则度量候选文档 D_i 与已选集合 S 中任一文档 D_j 的相似性。两者可采用相同的相似度度量（如余弦相似度），也可根据任务需求分别设计。

在信息检索的实际应用中，MMR 算法通常以如下方式执行：首先初始化一个空集合 S，并从初始结果集 R 中选取与查询 Q 相关性最高的文档作为 S 的首个元素。随后，通过迭代过程，对 $R \setminus S$ 中的每个剩余候选文档，依据公式 7.12 计算其 MMR 分数——该分数综合了其与查询的相关性以及与已选文档的最大相似度（即冗余程度）。每次迭代选择 MMR 分数最高的文档加入 S，直至 S 达到预设的规模。最终得到的结果集在保持高相关性的同时，显著降低了内容冗余。

基于上述流程，信息检索场景下的 MMR 算法核心实现如代码清单 7.6 所示：

代码 7.6: 基于 MMR 算法的信息检索代码

```
import numpy as np
from sklearn.metrics.pairwise import cosine_similarity
from sklearn.feature_extraction.text import TfidfVectorizer

def mmr_selection(query: str, documents: list, n_selected: int = 5, lambda_param: float = 0.5, ↩
    ↪ vectorizer=None) -> list:
    # 初始化文本向量化工具
    if vectorizer is None:
        vectorizer = TfidfVectorizer(stop_words='english') # 英文停用词处理
    # 合并查询和文档，统一向量化（确保向量空间一致）
    all_texts = [query] + documents
    tfidf_matrix = vectorizer.fit_transform(all_texts)
    # 提取查询向量和文档向量
```

```
    query_vec = tfidf_matrix[0:1] # 第0个是查询向量
    doc_vecs = tfidf_matrix[1:] # 后续是文档向量（与documents顺序对应）
    # 计算所有文档与查询的相关性（余弦相似度）
    relevance = cosine_similarity(doc_vecs, query_vec).flatten() # 形状：(n_docs,)
    # 初始化已选集合（S）和待选集合（剩余文档索引）
    selected_indices = []
    remaining_indices = list(range(len(documents)))
    # 第一次选择：仅按相关性（因为S为空，无法计算多样性）
    if remaining_indices:
        first_idx = np.argmax(relevance[remaining_indices])
        first_idx = remaining_indices[first_idx] # 转换为原始索引
        selected_indices.append(first_idx)
        remaining_indices.remove(first_idx)
    # 迭代选择：平衡相关性和多样性
    for _ in range(n_selected - 1):
        if not remaining_indices:
            break # 待选集合为空时停止
        mmr_scores = []
        for doc_idx in remaining_indices:
            # 1. 计算当前文档与查询的相关性（已归一化）
            rel_score = relevance[doc_idx]
            # 2. 计算当前文档与已选文档的最大相似度（多样性指标）
            # 已选文档向量
            selected_vecs = doc_vecs[selected_indices]
            # 当前文档与所有已选文档的相似度
            sim_to_selected = cosine_similarity(doc_vecs[doc_idx:doc_idx+1], selected_vecs).flatten↩
                ↪ ()
            max_sim = np.max(sim_to_selected) # 取最大相似度（最冗余的已选文档）
            # 3. 计算MMR分数：*相关性 - (1-)*最大冗余度
            mmr_score = lambda_param * rel_score - (1 - lambda_param) * max_sim
            mmr_scores.append(mmr_score)
        # 选择MMR分数最高的文档
        best_idx = remaining_indices[np.argmax(mmr_scores)]
        selected_indices.append(best_idx)
        remaining_indices.remove(best_idx)
    # 返回筛选后的文档（按选择顺序）
    return [documents[i] for i in selected_indices]
```

为验证上述实现，可使用构造的样例数据进行测试。代码清单 7.7 提供了一个完整的调用示例，展示了如何利用 MMR 从一组与查询相关的文档中筛选出低冗余、高相关性的结果子集：

代码 7.7: 基于 MMR 算法的信息检索代码 (程序入口)

```
if __name__ == "__main__":
    # 示例数据：与"人工智能在医疗中的应用"相关的文档
    query = "人工智能 医疗 应用"
    documents = [
        "人工智能可用于医学影像识别，辅助医生检测肿瘤。",
        "医学影像识别是人工智能在医疗中的重要应用，尤其擅长肺癌筛查。", # 与第1篇高度相似
```

```
    "AI技术能分析电子病历，预测患者的疾病风险。",
    "电子病历分析可帮助医生快速判断患者病情，AI在其中起到关键作用。", # 与第3篇高度相似
    "人工智能驱动的药物研发能缩短新药上市时间。",
    "AI在手术机器人中的应用提高了微创手术的精度。",
    "手术机器人结合AI技术后，患者术后恢复时间显著缩短。" # 与第6篇高度相似
]

# 用MMR筛选3篇文档（=0.6：稍侧重相关性）
selected = mmr_selection(
    query=query,
    documents=documents,
    n_selected=3,
    lambda_param=0.6,
    # 中文需使用适合的向量器（这里用默认参数，实际可优化）
    vectorizer=TfidfVectorizer(analyzer='word', stop_words=['的', '是', '在'])
)

# 输出结果
print("筛选后的文档：")
for i, doc in enumerate(selected, 1):
    print(f"{i}. {doc}")
```

在推荐系统场景中，MMR 同样适用于重排序阶段的多样性控制。此时，系统通常已通过精排或重排模型获得了 top-K 个候选物品及其对应的打分。这些打分可直接作为公式 7.12 中的第一项 $\mathrm{Sim}_1(D_i, Q)$，即物品与用户意图（或上下文）的相关性度量。而第二项物品间的相似度 $\mathrm{Sim}_2(D_i, D_j)$，则可借助推荐模型中物品 ID 对应的 embedding 向量，通过点积或余弦相似度进行计算。

值得注意的是，若将多样性打散机制前置至精排模块，也可将 MMR 视为一种贪心策略：从 top-N 候选中逐步选择 M（$M < N$）个物品，使得最终集合在相关性与多样性之间取得平衡。尽管如此，**重排序阶段仍是实现多样性控制的关键环节**，因其能基于更完整的上下文信息进行全局优化。当然，也可在前链路模块中进行初步尝试，形成多阶段协同的多样性保障机制。

7.5.2 DPP 算法

DPP（Determinantal Point Process，行列式点过程）的思想最早可追溯至 1975 年 Macchi 在论文《The coincidence approach to stochastic point processes》[10] 中的工作。该模型最初用于描述量子力学中费米子的统计行为。由于泡利不相容原理，费米子倾向于彼此“排斥”，从而在空间中呈现出天然的多样性分布。此后，DPP 在数学领域得到了深入发展，被广泛应用于随机矩阵特征值分布、随机生成树、不相交路径等问题的研究。

DPP 从数学与物理领域向机器学习领域的迁移始于 2011 年。Kulesza 与 Taskar 在 UAI 会议上发表了题为《Learning Determinantal Point Processes》[8] 的开创性论文，首次提出了**可学习的、带参数化的 DPP 模型**，使得 DPP 能够从数据中自动拟合多样性偏好。随后，他们在 2012 年于 *Foundations and Trends in Machine Learning* 上发表了综述性文章《Determinantal Point Processes for Machine Learning》[7]，系统阐述了 DPP 的直观含义、核心算法及其在机器学习中的典型应用场景。这些工作极大地推动了 DPP 在多样性推荐、文档摘要、图像检索等任务中的应用。

在机器学习任务中，DPP 常被用作一种建模元素间“排斥性”（repulsion）的概率模型，特别适用于需要在保持相关性的同时增强结果多样性的场景。例如，在信息检索或推荐系统中，DPP 能够有效刻画被选物品之间的不相似性，从而避免冗余结果的出现。然而，从贝叶斯推断的角度来看，DPP 的最大后验概率（MAP, Maximum A Posteriori）推断是一个 NP-难问题。传统的精确求解方法计算复杂度通常高达 $O(N^4)$，严重制约了其在大规模

推荐场景中的实际部署。虽然后续研究提出了复杂度为 $O(N^3)$ 的近似算法，但往往以牺牲解的质量为代价：即在降低计算开销的同时，难以保证多样性效果的稳定性与最优性。

针对这一挑战，Chen 等人在 NeurIPS 2018 发表的论文《Fast Greedy MAP Inference for Determinantal Point Process to Improve Recommendation Diversity》[2] 提出了一种时间复杂度为 $O(N^3)$ 高效且高精度的贪心 MAP 推断算法。该方法在显著降低计算复杂度的同时，有效保持了 DPP 所期望的多样性增益，从而为 DPP 在工业级推荐系统中的落地提供了切实可行的技术路径。

下面我们重点介绍 DPP 的数学原理和几何解释。DPP 是一种优雅的概率模型，能够表达负相关关系。形式上，给定离散集合 $Z = \{1, 2, \cdots, N\}$，则 DPP $\mathcal{P}$ 是作用在 2^Z(即集合 Z 的所有子集) 上的概率测度。当 $\mathcal{P}$ 对空集赋予非零概率时，存在一个半正定矩阵 $\left(\mathbf{L} = \mathbf{B}^T\mathbf{B}\right) \in \mathbb{R}^{N\times N}$，使得对于每个子集 $Y \subseteq Z$ 有：

$$\mathcal{P}(Y) \propto \det(\mathbf{L}_Y) = \text{Vol}^2\left(\{\mathbf{B}_i\}_{i\in Y}\right) \tag{7.13}$$

注意这里对于矩阵 $\mathbf{L}$ 增加了半正定的限制，这种 DPP 定义实际上被称之为“L-ensemble”的 DPP [7]。

从公式7.13可以看出，一个集合 Y 的 DPP 概率与集合中元素组成的集合 $\{\mathbf{B}_i\}_{i\in Y}$ 中所有向量 $\mathbf{B}_i$ 张成的高维超平行体的体积平方成正比。因此，当集合 Y 对应的所有向量 $\mathbf{B}_i$ 张成的高维超平行体的体积越大，则在 DPP 中对应的概率也会越大。图7.5展示了二维空间和三维空间中的向量张成的几何结构，分别对应的是平行四边形和平行六面体。如果每个向量的长度不变，任意两个向量之间的夹角变小，都会导致这两个向量的相似性增大，从而导致超几何体的体积变小，DPP 的概率也会变小。

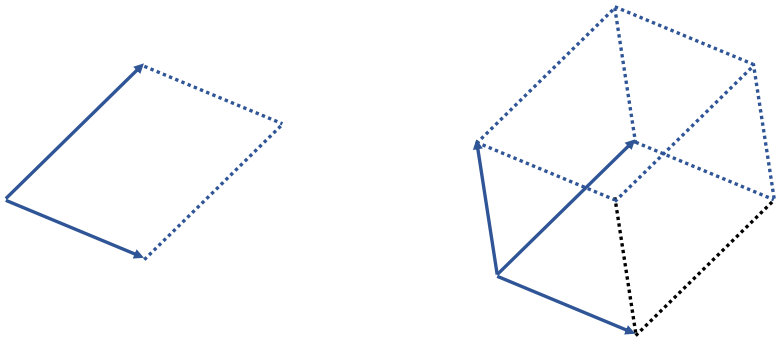

图 7.5: DPP 几何结构示意图。

公式7.13所示的 DPP 对应的 MAP 推断公式如下：

$$\mathcal{P}(Y)_{\text{MAP}} = \arg\max_{Y\subseteq Z} \det(\mathbf{L}_Y) \tag{7.14}$$

在推荐场景中，我们可以将矩阵 $\mathbf{B}$ 中每个物品对应的向量 $\mathbf{B}_i$ 定义为 $\mathbf{B}_i = r_i\mathbf{f}_i$，其中 r_i 是这个物品对应的 score，具体可以用推荐系统当前模块下物品的融合分。而 $\mathbf{f}_i$ 是物品 i 对应的一个向量，可以是重排模型或者精排模型产出的 item embedding。注意这里的向量 $\mathbf{f}_i$ 需要做归一化的处理，保证向量模长 $\|\mathbf{f}_i\|_2 = 1$。因此矩阵 $\mathbf{L}$ 中对应的元素可以写作：

$$\mathbf{L}_{ij} = \langle \mathbf{B}_i, \mathbf{B}_j\rangle = \langle r_i\mathbf{f}_i, r_j\mathbf{f}_j\rangle = r_i r_j\langle \mathbf{f}_i, \mathbf{f}_j\rangle \tag{7.15}$$

所以 kernel 矩阵 $\mathbf{L}$ 可以写成：

$$\mathbf{L} = \text{Diag}(\mathbf{r}) \cdot \mathbf{S} \cdot \text{Diag}(\mathbf{r}) \tag{7.16}$$

注意这里的 $\mathbf{S}$ 是一个相似度矩阵且 $\mathbf{S}_{ij} = \langle \mathbf{f}_i, \mathbf{f}_j\rangle$，$\text{Diag}(\mathbf{r})$ 是根据所有物品的相关性分数组成的相关性向量 $\mathbf{r}$ 所构成的对角矩阵。由于 $\langle \mathbf{f}_i, \mathbf{f}_j\rangle$ 实际上表示了 cosine 相似度，取值范围是 $[-1, 1]$。为了保证 $\mathbf{L}$ 是半正定矩阵，可以通过 $\mathbf{S}_{ij} = \frac{1+\langle \mathbf{f}_i, \mathbf{f}_j\rangle}{2}$ 的处理方式将 $\mathbf{S}_{ij}$ 取值限制在 $[0, 1]$ 范围内。

根据行列式的如下两个性质：

1. $\det(AB) = \det(A)\det(B)$
2. 对角矩阵的行列式等于对角元素的乘积：$\det(\text{Diag}(\mathbf{r})) = \prod_{i=1}^{N} r_i$

我们可以得到矩阵 $\mathbf{L}$ 的对数行列式为：

$$\log\det \mathbf{L} = \sum_{i=1}^{N} \log(r_i^2) + \log\det(\mathbf{S}) \tag{7.17}$$

其中第一项 $\sum_{i=1}^{N} \log(r_i^2)$ 表示用户与子集 Y 中所有物品的相关性，而公式第二项 $\log\det(\mathbf{S})$ 表示整个子集 Y 中物品之间的多样性。

在推荐系统中为了更好地平衡上述相关性与多样性，可以在上式中增加 λ 超参数来进行权衡得到：

$$\log\det \mathbf{L}' = \lambda \sum_{i=1}^{N} \log(r_i^2) + (1-\lambda)\log\det(\mathbf{S}) \tag{7.18}$$

其中矩阵 $\mathbf{L}' = \text{Diag}(\exp(\alpha\mathbf{r})) \cdot \mathbf{S} \cdot \text{Diag}(\exp(\alpha\mathbf{r}))$，$\alpha = \theta/(2(1-\theta))$。

对于公式7.17和7.18的最大化求解，文献 [4] 从理论上证明了 DPP 的 MAP 推理是一种子模函数最大化 (submodular function) 问题，而子模函数最大化问题虽然是 NP-难问题，但通常有贪心的迭代算法进行近似的求解。假设集合函数 (set function)f 是定义在集合上 2^Z 上的实值函数。如果一个集合函数 f 的**边际增益 (marginal gains) 是非递增的**，即对于任意的 $i \in Z$ 和任意满足 $X \subseteq Y \subseteq Z$ $\{i\}$ 的集合，都有如下公式成立：

$$f(X \cup \{i\}) - f(X) \geq f(Y \cup \{i\}) - f(Y) \tag{7.19}$$

而贪心求解 DPP 对应的子模函数最大化问题则是首先初始化目标集合 $Y = \emptyset$，然后在每次迭代中根据如下公式7.20更新，直到最终目标集合中物品数量达到预期个数终止。

$$j = \arg\max_{i \in Z \setminus Y} \log\det(\mathbf{L}_{Y\cup\{i\}}) - \log\det(\mathbf{L}_Y) \tag{7.20}$$

对于公式7.20的求解，该论文中引入了 Cholesky 分解的算法，每次迭代根据 $\mathbf{L}_Y$ 的 Cholesky 分解结果 $\mathbf{L}_Y = \mathbf{V}\mathbf{V}^T$，通过递推公式计算出 $\mathbf{L}_{Y\cup\{i\}}$，进而得到 $\log\det(\mathbf{L}_{Y\cup\{i\}}) - \log\det(\mathbf{L}_Y)$ 的结果，从而求出满足最大条件的物品编号 j。对于任意的物品 $i \in Z \setminus Y$，矩阵 $\mathbf{L}_{Y\cup\{i\}}$ 的 Cholesky 分解可以通过如下递推公式进行更新：

$$\mathbf{L}_{Y\cup\{i\}} = \begin{bmatrix} \mathbf{L}_Y & \mathbf{L}_{Y,i} \\ \mathbf{L}_{i,Y} & \mathbf{L}_{ii} \end{bmatrix} = \begin{bmatrix} \boldsymbol{V} & \mathbf{0} \\ \boldsymbol{c}_i & d_i \end{bmatrix} \begin{bmatrix} \boldsymbol{V}^T & \boldsymbol{c}_i^T \\ \mathbf{0}^T & d_i \end{bmatrix} \tag{7.21}$$

其中 $\mathbf{L}_{Y,i}$ 表示的是取矩阵 $\mathbf{L}$ 中的 Y 中包含的编号对应的行以及第 i 列形成的子矩阵。$\mathbf{L}_{i,i}$ 表示矩阵 $\mathbf{L}$ 中位置 (i,j) 的元素。

根据上面的 Cholesky 分解公式，通过矩阵乘法可得 $\mathbf{L}_{Y,i}$ 和 $\mathbf{L}_{ii}$ 与待求解的 $\boldsymbol{c}_i$ 和 d_i 之间的关系：

$$\begin{aligned} \boldsymbol{V}\boldsymbol{c}_i^\top &= \mathbf{L}_{Y,i} \\ d_i^2 &= \mathbf{L}_{ii} - \|\boldsymbol{c}_i\|_2^2 \end{aligned} \tag{7.22}$$

由于公式7.21中 $\mathbf{L}_{Y\cup\{i\}}$ 也可以写成一个矩阵与其转置相乘的形式，因此 $\mathbf{L}_{Y\cup\{i\}}$ 的行列式等于右侧分解矩阵行列式的平方。由此可得：

$$\det(\mathbf{L}_{Y\cup\{i\}}) = \det(\boldsymbol{V}\boldsymbol{V}^\top) \cdot d_i^2 = \det(\mathbf{L}_Y) \cdot d_i^2 \tag{7.23}$$

这样贪心迭代的目标函数可以转化为：

$$\begin{aligned} j &= \arg\max_{i \in Z \setminus Y} \log\det(\mathbf{L}_{Y\cup\{i\}}) - \log\det(\mathbf{L}_Y) \\ &= \arg\max_{i \in Z \setminus Y} \log\det(\mathbf{L}_Y) + \log(d_i^2) - \log\det(\mathbf{L}_Y) \\ &= \arg\max_{i \in Z \setminus Y} \log(d_i^2) \end{aligned} \tag{7.24}$$

在一次迭代完成求出最优的 j 之后，就可以更新公式7.21中的矩阵 $\mathbf{L}_{Y\cup\{j\}}$ 为：

$$\mathbf{L}_{Y\cup\{j\}} = \begin{bmatrix} \boldsymbol{V} & \mathbf{0} \\ \boldsymbol{c}_j & d_j \end{bmatrix} \begin{bmatrix} \boldsymbol{V} & \mathbf{0} \\ \boldsymbol{c}_j & d_j \end{bmatrix}^\top \tag{7.25}$$

注意这里矩阵 $\mathbf{L}_{Y\cup\{j\}}$ 的迭代公式与集合 Y 密切相关，因此 $\mathbf{L}_Y$ 中每一行分别与加入集合 Y 的物品实际编号相对应，其中第一行表示的是相关性最大的那个物品对应的向量。

接着可以构造出新的 $\boldsymbol{c}_i'$ 满足：

$$\begin{bmatrix} \boldsymbol{V} & \mathbf{0} \\ \boldsymbol{c}_j & d_j \end{bmatrix} \boldsymbol{c}_i'^\top = \mathbf{L}_{Y\cup\{j\},i} = \begin{bmatrix} \mathbf{L}_{Y,i} \\ \mathbf{L}_{ji} \end{bmatrix} \tag{7.26}$$

结合公式7.22中关于 $\boldsymbol{c}_i$ 的等式，即可得到向量 $\boldsymbol{c}$ 的迭代公式：

$$\boldsymbol{c}_i' = \begin{bmatrix} \boldsymbol{c}_i & \dfrac{\mathbf{L}_{ji} - \langle \boldsymbol{c}_j, \boldsymbol{c}_i \rangle}{d_j} \end{bmatrix} \doteq [\boldsymbol{c}_i \quad e_i] \tag{7.27}$$

此外，结合公式7.22可得数值 d_i 的迭代公式如下：

$$d_i'^2 = \mathbf{L}_{ii} - \|\boldsymbol{c}_i'\|_2^2 = \mathbf{L}_{ii} - \|\boldsymbol{c}_i\|_2^2 - e_i^2 = d_i^2 - e_i^2 \tag{7.28}$$

根据上述的公式推导，我们可以给出对应的 python 代码如下所示：

代码 7.8: 基于 DPP 贪心算法的多样性打散代码

```python
import numpy as np
def fast_greedy_map_inference(item_embeddings, r_u, theta, K):
    """
    参数:
        item_embeddings: 物品嵌入矩阵（二维numpy数组，形状为[N, D]，N为物品数，D为嵌入维度）
        r_u: 物品与用户的相关性得分数组（一维numpy数组，长度为N）
        theta: 平衡参数（0 theta 1），控制相关性与多样性权重
        K: 需选中的物品数量（终止条件）
    返回:
        Y_g: 选中的物品索引列表（长度为K）
    """
    N = item_embeddings.shape[0] # 物品总数
    if K <= 0 or K > N:
        raise ValueError(f"K必须为正整数且不超过物品总数{N}")

    # 1. 计算余弦相似度矩阵S并映射到[0,1]
    # 对嵌入向量归一化（余弦相似度 = 归一化后的点积）
    norm_embeddings = item_embeddings / np.linalg.norm(item_embeddings, axis=1, keepdims=True)
    # 计算余弦相似度矩阵（点积）
    cos_sim = norm_embeddings @ norm_embeddings.T # 形状[M, M]
    # 映射到[0,1]范围: (1 + cos_sim) / 2（余弦相似度本身范围为[-1,1]）
    S = (1 + cos_sim) / 2

    # 2. 构建核矩阵L'
    if theta == 1:
        alpha = 0 # 极端情况：只考虑相关性，不考虑多样性
    else:
        alpha = theta / (2 * (1 - theta)) # 平衡参数
    exp_alpha_r = np.exp(alpha * r_u) # 形状[N]
    # L' = Diag(exp(r_u)) · S · Diag(exp(r_u))
```

```
    L = np.diag(exp_alpha_r) @ S @ np.diag(exp_alpha_r) # 形状[N, N]

    # 3. 核心迭代过程
    # 初始化辅助参数
    c = [[] for _ in range(N)] # c[i]为物品i的辅助向量（列表存储，动态增长）
    d_squared = L.diagonal().copy() # d_i²初始为L的对角线元素，形状[N]
    Y_g = [] # 选中的物品集合

    while len(Y_g) < K:
        # 选择当前最优物品（未选中且log(d_i²)最大）
        valid_mask = np.ones(N, dtype=bool)
        valid_mask[Y_g] = False # 排除已选中物品
        valid_indices = np.where(valid_mask)[0]
        if len(valid_indices) == 0:
            break # 理论上不会触发，因K  N
        # 找到最大log(d_i²)对应的物品
        j = valid_indices[np.argmax(np.log(d_squared[valid_indices]))]
        Y_g.append(j)

        # 若已选满K个，提前终止
        if len(Y_g) == K:
            break
        # 增量更新所有未选中物品的c[i]和d_squared[i]
        j_idx = len(Y_g) - 1 # 当前j是第j_idx+1个选中的物品（0-based）
        d_j = np.sqrt(d_squared[j]) # d_j = sqrt(d_j²)

        for i in range(N):
            if i not in Y_g:
                # 计算内积<c_j, c_i>（c_j是已选物品j的辅助向量）
                dot_product = np.dot(c[j], c[i]) if (len(c[j]) > 0 and len(c[i]) > 0) else 0.0
                # 计算e_i = (L_ji - 内积) / d_j
                e_i = (L[j, i] - dot_product) / d_j
                # 更新c[i]：追加e_i
                c[i].append(e_i)
                # 更新d_i²：d_i² -= e_i²

                d_squared[i] -= e_i **2

    return Y_g
```

由于 DPP 贪心算法使用了向量和数值迭代的公式来更新 $\mathbf{c}_i$ 和 d_i，在第 k 次迭代时，每个物品的向量 $\mathbf{c}_i$ 记录了该物品与前 k 个已选物品的相关性，因此 $\mathbf{c}_i$ 的长度为 k，更新该向量的计算复杂度为 $O(k)$。由于未选择的物品有 $N-k$ 个，一次迭代的计算复杂度为 $O(k(N-k))$。假设算法总共迭代 K 次，则总的计算复杂度为 $\sum_{k=1}^{K} O(k(N-k)) = O(K^2N)$。因此，如果没有任何约束而选择全部 N 个物品，则总计算复杂度为 $O(N^3)$；而如果仅选取 $K \ll N$ 个物品，则计算复杂度可显著降低至 $O(K^2N)$，这使其在实际推荐系统中具有良好的可扩展性。

7.5.3 SSD 算法

SSD（Sliding Spectrum Decomposition，滑动频谱分解）算法源自小红书团队在 KDD 2021 会议上发表的论文《Sliding Spectrum Decomposition for Diversified Recommendation》。该算法的提出源于对信息流推荐场景中用户行为特性的深入观察：在用户持续下滑浏览内容的过程中，其内容消费深度往往较深，且对整个浏览序列的多样性具有**时序连贯性感知**，即当前内容的多样性体验不仅受当前物品影响，也受到历史已浏览内容的累积影响。传统基于滑动窗口的多样性方法（如窗口内 DPP）通常仅考虑局部窗口内的物品，而将窗口外的历史信息直接丢弃，这不仅容易陷入局部最优，也难以捕捉长序列上下文对整体多样性感知的贡献。

为克服这一局限，SSD 算法将用户完整的长度为 T 的浏览序列全部纳入建模范围。假设滑动窗口长度为 W，则整个序列可划分为 $L = TW + 1$ 个连续滑动窗口。若每个物品的 Embedding 向量维度为 d，则可将所有窗口的物品 Embedding 矩阵沿窗口维度堆叠，构成一个三维张量 $\mathcal{X} \in \mathbb{R}^{L \times W \times d}$，如下图7.6所示。

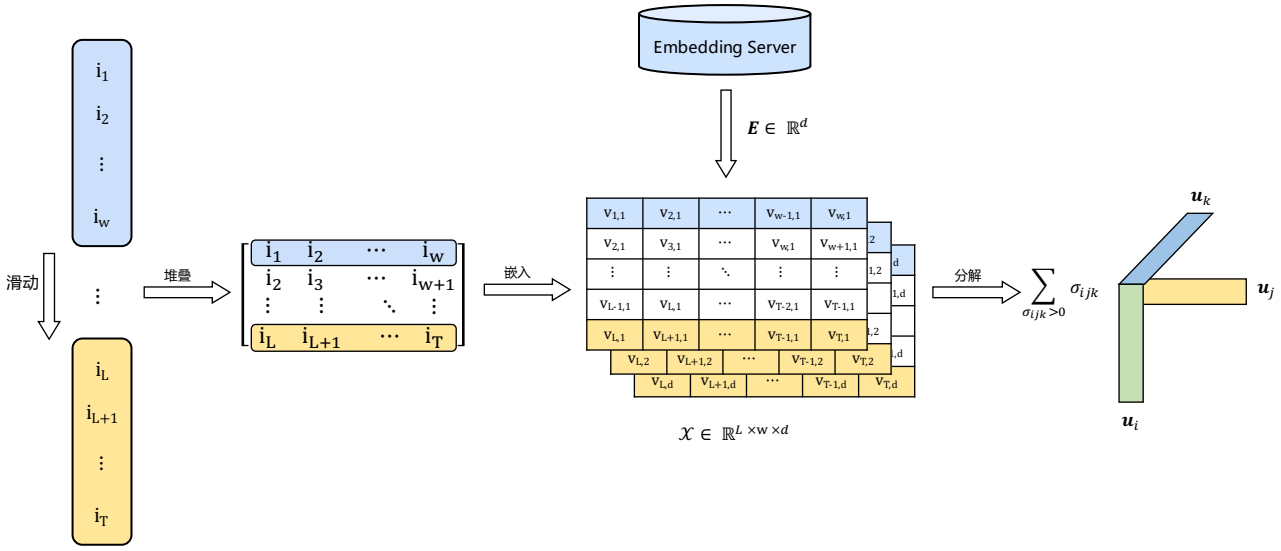

图 7.6: SSD 打散示意图。

受传统信号处理中奇异谱分析（Singular Spectrum Analysis, SSA）的启发，SSD 算法采用高阶奇异值分解（Higher-Order Singular Value Decomposition，Higher-Order SVD）对张量 $\mathcal{X}$ 进行分解，其形式可表示为：

$$X = \sum_{\sigma_{ijk}>0} \sigma_{ijk}\, \mathbf{u}_i^{(1)} \otimes \mathbf{u}_j^{(2)} \otimes \mathbf{u}_k^{(3)} \tag{7.29}$$

其中 σ_{ijk} 表示非零奇异值，$\mathbf{u}_i^{(1)} \in \mathbb{R}^L$、$\mathbf{u}_j^{(2)} \in \mathbb{R}^W$、$\mathbf{u}_k^{(3)} \in \mathbb{R}^d$ 分别为沿序列、窗口和特征三个维度的正交基向量，，$\otimes$ 表示向量外积。

与 DPP 中“体积”反映多样性的思想类似，SSD 算法将张量 $\mathcal{X}$ 的“序列体积”定义为其所有非零奇异值的乘积（或加权和），以此度量整个滑动窗口序列中物品间的全局多样性。在实际优化中，同样需要在相关性与多样性之间进行权衡，目标函数形式如下：

$$\max_{\{i_1,\ldots,i_T\} \subset Z} \sum_{t=1}^{T} r_{i_t} + \gamma \sum_{\sigma_{ijk}>0} \sigma_{ijk} \tag{7.30}$$

其中 r_{i_t} 为物品 i_t 的相关性得分，γ 为控制多样性强度的超参数。

在求解该优化问题时，SSD 依然采用贪心策略，但其核心创新在于引入了一种计算高效且数值稳定的正交化机制：**一步改进型格拉姆–施密特正交化**（one-step Modified Gram-Schmidt, one-step MGS）。为清晰理解这一机制的动机与设计，有必要首先回顾传统 Gram-Schmidt（GS）正交化及其改进版本 MGS 的完整流程。

传统 Gram-Schmidt（GS）正交化用于将一组线性无关的向量转换为正交向量组，其核心步骤如下：

1. 给定待正交化的向量集合 $\{\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_k\}$。
2. 初始化第一个正交向量 $\boldsymbol{u}_1 = \boldsymbol{v}_1$，并归一化得到单位向量 $\boldsymbol{e}_1 = \boldsymbol{u}_1 / \|\boldsymbol{u}_1\|$。
3. 对第 m 个向量 $\boldsymbol{v}_m$（其中 $m \geq 2$），扣除其在所有已构造的正交单位向量 $\{\boldsymbol{e}_1, \ldots, \boldsymbol{e}_{m-1}\}$ 上的投影：
$$\boldsymbol{u}_m = \boldsymbol{v}_m - \sum_{p=1}^{m-1} (\boldsymbol{v}_m^\top \boldsymbol{e}_p) \boldsymbol{e}_p \tag{7.31}$$
4. 将 $\boldsymbol{u}_m$ 归一化得到 $\boldsymbol{e}_m = \boldsymbol{u}_m / \|\boldsymbol{u}_m\|$，重复步骤 3 和 4，直至所有向量完成正交化。

尽管 GS 在理论上能生成正交基，但在浮点运算中，由于舍入误差的累积，实际得到的向量往往仅近似正交，尤其在向量数量较多或条件数较差时，正交性会显著退化。

为提升数值稳定性，**改进型 Gram-Schmidt**（Modified Gram-Schmidt, MGS）对算法流程进行了关键调整：不再一次性扣除所有历史投影，而是采用**逐步正交化**策略。其具体步骤如下：

1. 给定待正交化的向量集合 $Z = \{\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_k\}$。
2. 初始化迭代步数 $k = 1$，令第一个正交向量 $\boldsymbol{u}_1 = \boldsymbol{v}_1$，归一化得 $\boldsymbol{e}_1 = \boldsymbol{u}_1 / \|\boldsymbol{u}_1\|$。
3. 对于第 $k \geq 2$ 次迭代，对所有尚未正交化的向量（即 $\boldsymbol{v}_k, \boldsymbol{v}_{k+1}, \ldots, \boldsymbol{v}_n$），依次扣除其在当前新生成的正交向量 $\boldsymbol{e}_{k-1}$ 上的投影：
$$\boldsymbol{v}_j \leftarrow \boldsymbol{v}_j - (\boldsymbol{v}_j^\top \boldsymbol{e}_{k-1}) \boldsymbol{e}_{k-1}, \quad \forall j \geq k \tag{7.32}$$
这一操作在每一步后立即更新剩余向量，从而将误差限制在局部，避免向后传播。
4. 将更新后的 $\boldsymbol{v}_k$ 作为 $\boldsymbol{u}_k$，归一化得 $\boldsymbol{e}_k = \boldsymbol{u}_k / \|\boldsymbol{u}_k\|$，重复步骤 3 和 4，直至所有向量处理完毕。

MGS 的优势在于：每次正交化操作仅依赖当前已构造的正交向量，并立即作用于后续向量，因此舍入误差不会在后续步骤中被放大，数值稳定性显著优于传统 GS。此外，MGS 在处理线性相关向量时，部分向量在正交化过程中模长会逐渐趋于零，此时可通过设定阈值直接丢弃这些冗余向量，避免无效计算。

SSD 算法在此基础上进一步简化，提出**“一步改进型 MGS”**（one-step MGS）策略，以适配推荐系统中贪心选择的时序特性。具体而言，在每次迭代选择新物品时，SSD 并不对候选向量执行完整的 MGS 正交化，而是仅将其 Embedding 向量对**最新加入序列的物品向量**进行一次投影扣除。假设当前已选物品集合为 $\{i_1, \ldots, i_{t-1}\}$，则对任一未选候选物品 j，其 Embedding 向量 $\boldsymbol{v}_j$ 被更新为：
$$\boldsymbol{v}_j \leftarrow \boldsymbol{v}_j - \frac{\langle \boldsymbol{v}_j, \boldsymbol{v}_{i_{t-1}} \rangle}{\|\boldsymbol{v}_{i_{t-1}}\|^2} \boldsymbol{v}_{i_{t-1}} \tag{7.33}$$
即仅扣除其在最新选中物品向量上的分量。这种“增量式”近似虽未实现全局正交，但能有效削弱与最近内容的相似性，契合用户对**局部时序多样性**的感知，同时大幅降低计算开销。

基于上述思想，SSD 算法的 Python 实现如下所示：

代码 7.9: 基于 SSD 算法的多样性打散代码

```python
import numpy as np

def ssd_without_sliding_window(r, embeddings, T, gamma):
    """
    参数:
        r: 物品的相关性得分列表，长度为N
        embeddings: 物品的嵌入向量矩阵，形状为[N, d]（N为物品数，d为嵌入维度）
        T: 推荐序列的长度
        gamma: 相关性与多样性的权衡系数

    返回:
        选中的物品序列索引列表，长度为T
    """
    N, d = embeddings.shape
```

```
    selected = [] # 存储选中的物品索引
    # 初始化：选择第一个相关性得分最高的物品
    i_t = np.argmax(r)
    selected.append(i_t)
    # 初始化体积V（Volume）
    V = np.linalg.norm(embeddings[i_t])

    for t in range(1, T): # 循环选择剩余T-1个物品
        # 对所有未选中的物品，执行一步MGS正交化
        for j in range(N):
            if j not in selected:
                # 计算内积：v_j与已选物品i_t的嵌入向量的内积
                dot_product = np.dot(embeddings[j], embeddings[i_t])
                # 计算分母：已选物品i_t的嵌入向量的自内积（范数平方）
                denominator = np.dot(embeddings[i_t], embeddings[i_t])
                # 执行一步MGS：v_j = v_j - (v_j · v_{i_t}/v_{i_t} · v_{i_t}) * v_{i_t}
                embeddings[j] = embeddings[j] - (dot_product / denominator) * embeddings[i_t]

        # 选择下一个最优物品：最大化 r_j + ||v_j|| * V
        candidates = [j for j in range(N) if j not in selected]
        scores = [r[j] + np.linalg.norm(embeddings[j]) * gamma * V for j in candidates]
        i_t = candidates[np.argmax(scores)]
        selected.append(i_t)
        # 更新体积V
        V = np.linalg.norm(embeddings[i_t]) * V
    return selected
```

值得注意的是，one-step MGS 的关键优势在于其增量计算特性：无需显式维护完整的正交基，也无需存储所有历史 Embedding，仅需在每次迭代中对候选向量执行一次投影操作。该操作的时间复杂度为 $O(d)$（d 为 Embedding 维度），远低于传统 MGS 的 $O(td)$（t 为已选物品数）。因此，SSD 贪心推理的整体时间复杂度为 $O(NTd)$，其中 N 为候选物品总数，T 为最终序列长度。在典型推荐场景中，T 通常仅为 20–50，而 N 可达数万量级，此时 SSD 算法的复杂度显著优于传统 DPP 贪心算法的 $O(N^2d)$，能够满足工业级实时推荐系统对低延迟与高吞吐的严苛要求，同时在多样性效果上保持竞争力。

7.6 重排后处理

在重排模块中，完成序列生成与序列评估之后，系统通常仍会引入一个独立的后处理（post-processing）阶段，对最终输出的推荐列表进行进一步的策略性调整。该阶段的核心目标是在尽可能不破坏重排模型排序结果的前提下，对推荐序列进行约束性修正与工程化增强，从而满足多样性控制、业务策略执行以及系统可观测性等多方面需求。

首先，在结果结构优化方面，重排后处理模块通常会对输出序列进行**多样性打散（diversity diversification）**操作。例如，在短视频或信息流场景中，系统可能需要避免同一作者、同一内容类型或高度相似内容的连续出现，从而降低用户的内容疲劳感，提升整体浏览体验。该类操作通常基于规则约束或轻量级贪心策略，在局部窗口内对排序结果进行微调，以实现内容结构上的平滑分布。

其次，在业务策略执行层面，后处理模块往往承担**保量策略（quota control）与强插策略（forced injection）**的落地职责。例如，在电商短视频或内容电商推荐场景中，系统可能需要在特定用户群体、特定流量刷次或特定时间窗口内，强制插入一定比例的商业内容、活动内容或运营内容，以满足平台商业化或运营目标。这类策

略通常以规则系统或策略引擎的形式实现，并与重排结果进行融合调整。需要指出的是，这类后处理机制与精排阶段的流量控制与强插逻辑具有一定相似性，但其作用位置更靠近最终曝光结果，因此对线上指标的影响更为直接。

在工程实现层面，重排后处理模块还承担重要的数据输出与系统对接功能。具体而言，系统通常会将重排过程中的关键中间信息进行持久化存储，例如写入 Redis 等在线缓存系统，以支持实时查询与策略回溯。同时，部分关键日志数据会通过 Kafka 等消息队列系统进行异步发送，用于离线分析、模型训练样本构建以及策略效果评估等下游任务。这一过程构成了推荐系统从在线决策到离线学习的数据闭环。

此外，为了保证重排策略的可观测性与可优化性，后处理阶段通常还会引入较为完善的监控与埋点体系。例如，对不同业务线的曝光占比、强插比例、多样性指标（如类别覆盖率、作者覆盖率）、以及关键策略命中率等进行细粒度监控。这些指标不仅用于实时监控系统健康状态，也为后续策略迭代与模型优化提供重要依据。在工业实践中，这类监控体系通常也与实验平台（A/B Testing System）紧密结合，以实现对不同重排策略效果的系统性评估。

总体而言，重排后处理虽然在算法层面相对轻量，但在工业级推荐系统中却承担着不可或缺的“策略落地与系统闭环”作用。它连接了模型输出与最终曝光结果，是保障推荐系统在复杂业务约束下稳定运行的重要的组成部分。

7.7 本章小结

本章围绕级联式推荐系统架构中的重排模块展开系统性介绍。作为精排之后的关键排序优化环节，重排模块的核心目标是在单物品打分结果的基础上，从**序列（list/slate）整体优化**的角度进一步提升推荐结果的质量与业务适配能力。

首先，本章从“为什么需要重排”这一基本问题出发，分析了精排阶段基于单点打分（pointwise ranking）所固有的局限性，指出其难以建模物品之间的相互关系以及序列级用户体验优化目标。在此基础上，进一步阐述了重排模块的必要性，并对工业界常见的两类重排范式进行了概述：一类是以“序列生成 + 序列评估”为核心的两阶段重排框架，另一类是基于生成式模型的单阶段重排方法。尽管后者近年来发展迅速，但在大规模工业系统中，两阶段框架仍然占据主流地位。

随后，本章详细介绍了两阶段重排框架的核心组成。其中，在**序列生成阶段**，重点讨论了基于多目标融合排序的贪心生成方法以及基于 beam search 的序列搜索策略，并从算法角度分析了其在搜索空间约束与计算效率之间的权衡。在**序列评估阶段**，进一步讨论了对候选序列进行整体打分与选择的常见方法，并从强化学习的视角对两阶段框架进行了统一解释：序列生成可类比为策略网络（Actor / Policy），负责产生候选序列；而序列评估则可视为价值函数（Critic / Value），用于对不同序列进行质量评估与选择。

在此基础上，本章进一步介绍了重排后处理模块的关键作用。在该阶段，系统通常在不显著破坏模型排序结果的前提下，对最终输出进行规则化与结构化修正。其中重点包括**多样性打散**与**业务策略强插**两类操作，并分别介绍了常见的多样性优化方法，如 MMR、DPP 以及 SSD 等算法。在分析其数学原理的同时，也给出了相应的可实现算法框架与 Python 实现思路。

除此之外，本章还对重排后处理中的工程化环节进行了说明，包括保量策略与强插策略的落地机制、关键字段的结构化存储、在线结果的 Redis 持久化、Kafka 异步日志投递，以及面向多业务维度的监控指标体系建设等。这些工程组件共同构成了重排模块从模型输出到线上服务闭环的重要支撑。总体而言，重排模块通过引入序列级优化视角，有效弥补了精排阶段单点建模的不足，使推荐系统能够在用户体验、多样性约束以及业务目标之间取得更优平衡。

下一章将介绍多业务混合推荐系统中的混排模块。该模块主要解决不同业务流量之间的全局协调与分配问题，通过在更高层级上进行流量融合与调度，实现多业务目标下的整体最优。

7.8 参考文献

[1] Jaime Carbonell and Jade Goldstein. “The use of MMR, diversity-based reranking for reordering documents and producing summaries”. In: *Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval*. 1998, pp. 335–336.

[2] Laming Chen, Guoxin Zhang, and Eric Zhou. “Fast greedy map inference for determinantal point process to improve recommendation diversity”. In: *Advances in neural information processing systems* 31 (2018).

[3] Markus Freitag and Yaser Al-Onaizan. “Beam search strategies for neural machine translation”. In: *arXiv preprint arXiv:1702.01806* (2017).

[4] Jennifer Gillenwater, Alex Kulesza, and Ben Taskar. “Near-optimal map inference for determinantal point processes”. In: *Advances in Neural Information Processing Systems* 25 (2012).

[5] Yanhua Huang et al. “Sliding spectrum decomposition for diversified recommendation”. In: *Proceedings of the 27th ACM SIGKDD conference on knowledge discovery & data mining*. 2021, pp. 3041–3049.

[6] Michael Janner, Qiyang Li, and Sergey Levine. “Offline reinforcement learning as one big sequence modeling problem”. In: *Advances in neural information processing systems* 34 (2021), pp. 1273–1286.

[7] Alex Kulesza, Ben Taskar, et al. “Determinantal point processes for machine learning”. In: *Foundations and Trends® in Machine Learning* 5.2–3 (2012), pp. 123–286.

[8] Alex Kulesza and Ben Taskar. “Learning determinantal point processes”. In: *Learning* 7 (2011), pp. 1–2011.

[9] Bruce P Lowerre and B Raj Reddy. “Harpy, a connected speech recognition system”. In: *The Journal of the Acoustical Society of America* 59.S1 (1976), S97–S97.

[10] Odile Macchi. “The coincidence approach to stochastic point processes”. In: *Advances in Applied Probability* 7.1 (1975), pp. 83–122.

[11] Clara Meister, Tim Vieira, and Ryan Cotterell. “If beam search is the answer, what was the question?” In: *arXiv preprint arXiv:2010.02650* (2020).

[12] Steven M Rubin and Raj Reddy. “The LOCUS Model of Search and its Use in Image Interpretation.” In: *IJCAI*. Vol. 2. 1977, pp. 590–595.

[13] Sam Wiseman and Alexander M Rush. “Sequence-to-sequence learning as beam-search optimization”. In: *arXiv preprint arXiv:1606.02960* (2016).