

目 录

一、 项目背景与意义.....	3
1.1 项目背景.....	3
1.2 项目意义.....	3
二、 功能模块设计.....	4
2.1 总体架构概述.....	4
2.2 区域热销商品 Top3 统计.....	5
2.2.1 需求分析.....	5
2.2.2 功能实现流程.....	5
2.2.3 代码实现.....	6
2.3 热门商品 Top10 统计.....	8
2.3.1 需求分析.....	8
2.3.2 功能实现流程.....	8
2.3.3 代码实现.....	9
2.4 商品转化率分析.....	11
2.4.1 需求分析.....	11
2.4.2 功能实现流程.....	11
2.4.3 代码实现.....	12
2.5 商品-用户关系矩阵构建.....	13
2.5.1 需求分析.....	13
2.5.2 功能实现流程.....	14
2.5.3 代码实现.....	14
三、 运行结果展示.....	16
3.1 区域热销商品 Top3 统计.....	16
3.2 热门商品 Top10 统计.....	17
3.3 商品转化率分析.....	17
3.4 商品-用户关系矩阵构建.....	18
四、 总结.....	19
4.1 项目总结.....	19
4.1 心得体会.....	19
附录.....	21
功能 1 完整代码.....	21
功能 2 完整代码.....	2
功能 3 完整代码.....	3
功能 4 完整代码.....	3

一、项目背景与意义

1.1 项目背景

当下各大电商平台每天都会产生数亿条用户行为记录，包括浏览、点击、下单、支付等全链路数据。在传统数据库上做批量分析既低效又难以扩展，Spark RDD 可并行、分布式地处理 TB 级日志，实现分钟级甚至秒级的离线分析，满足业务快速迭代的需求。

本项目基于电商平台的典型用户行为日志（user_visit_action.txt），并结合城市信息（city_info.txt）与商品信息（product_info.txt），旨在对海量点击数据进行清洗、关联与建模，以支撑后续的精准推荐与运营决策。

在本项目中，地域(city_info.txt) 与商品(product_info.txt)双维度洞察是连接宏观战略与微观运营的桥梁。通过将每条用户行为打上“城市→大区”标签，我们能够跳出千篇一律的“全国总量”思维，看清不同区域对同一商品在兴趣、需求和购买意愿上的差异。这样的洞察帮助我们在市场投放时既能因地制宜，又能在多个区域之间形成可复制的最佳实践。

在运营层面，不同地区的用户在浏览商品、加入购物车、最终下单支付的转化路径会有各自的瓶颈和触发点。对这些数据的处理，能够让我们能够准确定位哪些环节需要优化，相应地，运营团队可以针对性地设计本地化促销活动、优化供应链节点，既提升用户满意度，也降低了库存与履约成本。

从长远来看，这种双维度分析可以构建平台持续自我迭代的能力。借助数据驱动，我们能不断验证区域策略的成效，快速调整资源倾斜方向，并在不同区域之间相互借鉴推广成熟模式，最终实现精准化的市场覆盖与高效的资本回报。

1.2 项目意义

该项目的核心意义在于将零散的用户行为、地域信息和商品属性有机融合，构建起一套完整的数据分析和建模体系。它不仅让业务团队能够基于真实的用户触达和偏好进行科学决策，也推动了组织内部从传统报表向数据驱动思维的转型。

首先，在运营决策方面，通过“地区 - 商品”和“用户 - 商品”双向剖析，我们能够精准识别不同市场对不同品类和品牌的需求强度，避免盲目铺货或泛泛而作的全国性促销，将有限的营销预算投放到最具回报潜力的区域与人群。

其次，个性化推荐系统得到了坚实的数据支撑。基于海量点击行为可以构建用户画像，不仅让协同过滤模型拥有更加丰富的输入，还能与内容过滤或知识图谱结合，显著提升推荐准确度，让消费者在千人千面中快速发现感兴趣的商品，从而提高平台的点击率与转化率。

从技术视角来看，项目采用 Spark RDD 的分布式架构，具备天然的可扩展性与高并发处理能力，可无缝对接定时调度管道，将每日、每小时乃至分钟级的数据刷新变为可能；结合可视化看板与预警机制，一旦出现异常波动或潜力点，即可第一时间响应。对技术团队而言，这一端到端的实战项目不仅加深了对大数据生态的理解，也为未来向 Spark SQL、流处理、深度学习推荐等更高级方向扩展奠定了坚实基础。

更重要的是，这套解决方案为后续的智能化、自动化演进奠定了基础。借助这条成熟的流水线，组织能够更敏捷地响应市场变化，将新思路快速验证并推向生产，从而在激烈的竞争环境中保持持续创新与差异化优势。

二、功能模块设计

2.1 总体架构概述

本项目基于 Spark 技术完成，Apache Spark 是一个面向大规模数据处理的分布式计算框架，具有内存优先的执行引擎和丰富的生态组件（Spark SQL、Streaming、MLlib、GraphX 等），能够在集群环境下以极高的性能完成批量和流式任务。它支持多语言（Scala、Java、Python、R），并通过 DAG 调度和内存管理显著缩短了数据分析和机器学习的迭代周期，成为现代大数据平台的核心引擎。

在 Spark 内部，数据被抽象为弹性分布式数据集（RDD）。RDD 是 Spark 最底层的核心数据结构，它本质上是一组不可变的分区化记录集合。每个 RDD 维护了自己的“血统”，记录了从哪个或哪些父 RDD 经过哪些转换（map、

filter、join 等) 生成而来。正是因为这种血统追踪机制, Spark 能够在节点或分区丢失时, 沿着血统图自动重算所需的数据分区, 实现高效的容错, 而无需额外的数据复制。

RDD 的操作分为两类: Transformations 和 Actions。转换操作返回新的 RDD, 而行动操作才会触发实际的计算。Spark 会把整个转换链条“懒惰”地记录下来, 只有在遇到行动时, 才会一次性地构建物理执行计划, 并在集群上并行执行。这种惰性求值不仅避免了中间数据的重复计算, 还可以对多个转换进行统一优化, 比如管道化、窄依赖与宽依赖的调度策略选择。

RDD 的几个关键特性使其非常强大: 一是惰性求值, 只有在触发行动时才会执行计算; 二是血统 (lineage) 追踪, 当节点或分区故障时, 可以根据血统信息重算丢失的数据, 从而实现容错; 三是多种持久化级别 (内存、磁盘或两者结合), 让中间结果可以反复重用, 大幅提升迭代式算法的性能。虽然 Spark SQL 和 Dataset 提供了更高级别的优化和更简洁的接口, 但 RDD 依然在某些需要精细化分区控制、自定义序列化或日志级别处理的场景下不可替代。对于那些需要深度调优或处理非结构化数据的项目, 掌握 RDD 的核心思想和使用方法, 仍然是大数据开发者的必备技能。

2.2 区域热销商品 Top3 统计

2.2.1 需求分析

对于每个大区 (area), 输出该大区点击量最高的 3 个商品及其点击次数。通过这个功能, 快速洞察各地区的热销商品, 指导区域化营销策略, 同时避免全平台“热门榜”对地区差异的掩盖, 实现更精准的资源投放和用户运营。

2.2.2 功能实现流程

(1) 多表关联

这一步的核心在于把分散在三张表中的信息汇聚到点击记录上。首先, 我们以 cityId 为键, 将用户的点击行为与城市信息表做 join, 这样每一条点击就被打上了对应的“城市名”和“大区”标签; 接着, 以 productId 为键, 再次与商品信息表 join, 为每条记录补充上“商品名称”。通过这两次关联, 原本只包含日期、用户与商品 ID 的点击日志, 变成了一个富含上下文的记录——既知道在哪个大区, 也知道点击的是哪款商品。

(2) 分组聚合

则是统计各大区、各商品的点击量。我们把每条已经附带了大区 and 商品 ID 的记录映射成以 (area, productId) 为键、数值 1 为值的键值对，然后调用 `reduceByKey((x, y) => x + y)`，把同一个大区内同一商品的所有点击次数累加起来。这样，就能快速得到一张“每个大区每个商品点击总数”的中间表，为后续的排名提供基础数据。

(3) 局部 Top N

则是计算每个大区点击量最强的前三名商品。在拥有“(area, productId) → clickCount”结果后，我们先将其转换成 (area, Iterable[(productId, clickCount)]），也就是把所有关于同一大区的商品聚到一个集合里。然后在每个大区的集合内部，对商品点击数进行降序排序，并只取前 3 条。这样既可以保持并行执行的高效性，又避免了将全量数据拉到 Driver 上做一次大范围排序的昂贵开销。

2.2.3 代码实现

(1) 读取并解析基础数据

城市信息 (city_info.txt)：读取后得到 (cityId, (cityName, area))，把城市 ID 映射到对应的城市名和大区名。

```
val cityInfoRDD: RDD[(String, (String, String))] = sc.textFile("C://Users//wei_jingxuan//Desktop//city_info.txt")
  .map(line => {
    val parts = line.split("\t")
    val cityId = parts(0)
    val cityName = parts(1)
    val area = parts(2)
    (cityId, (cityName, area))
  })
```

商品信息 (product_info.txt)：读取后得到 (productId, productName)，把商品 ID 映射到具体的商品名称。

```
val productInfoRDD: RDD[(String, String)] = sc.textFile("C://Users//wei_jingxuan//Desktop//product_info.txt")
  .map(line => {
    val parts = line.split("\t")
    val productId = parts(0)
    val productName = parts(1)
    (productId, productName)
  })
```

用户行为 (user_visit_action.txt)：提取出点击行为，将每条点击记录映射为 (cityId, (date, userId, productId))。其中只保留 `click_product_id ≠ -1` 的记录。

```
val clickRDD: RDD[(String, (String, String, String))] = sc.textFile("C://Users//wei_jingxuan//Desktop//user_visit_action.txt")
  .map(line => {
    val parts = line.split("\t")
    val date = parts(0)
    val userId = parts(1)
    val clickProductId = parts(7)
    val cityId = parts(12)
    (date, userId, clickProductId, cityId)
  })
  .filter { case (_, _, productId, _) => productId != "-1" }
  .map { case (date, userId, productId, cityId) =>
    (cityId, (date, userId, productId))
  }
```

(2) 关联城市信息

用点击记录的 cityId 去 join 城市信息，得到 (cityId, ((date, userId, productId), (cityName, area)))。此时每条点击记录都带上了该用户所处的城市名和大区。

```
val clickWithCity: RDD[(String, ((String, String, String), (String, String)))] =
  clickRDD.join(cityInfoRDD)
```

(3) 关联商品信息

把上一步的 RDD 重映射为以 productId 为 key，再与商品信息 join，得到了 (productId, ((date, userId, cityName, area), productName))。此时每条记录同时包含了商品名称和所属大区。

```
val readyForProductJoin: RDD[(String, (String, String, String, String, String))] = clickWithCity.map {
  case (_, ((date, userId, productId), (cityName, area))) =>
    (productId, (date, userId, cityName, area))
}

val finalJoin: RDD[(String, ((String, String, String, String), String))] =
  readyForProductJoin.join(productInfoRDD)
```

(4) 按“地区 × 商品”统计点击量

从关联完毕的记录里，提取 (area, productId)，并为每次点击记一分。使用 reduceByKey 对每对 (area, productId) 累加，得到每个商品在每个大区的总点击次数。

```
val readyForProductJoin: RDD[(String, (String, String, String, String, String))] = clickWithCity.map {
  case (_, ((date, userId, productId), (cityName, area))) =>
    (productId, (date, userId, cityName, area))
}

val finalJoin: RDD[(String, ((String, String, String, String), String))] =
  readyForProductJoin.join(productInfoRDD)
```

(5) 对每个大区取 Top 3

将 (area, productId, count) 转换成 (area, Iterable[(productId, count)])，然后对每个大区内部按点击次数降序排序，取前三条。

```

val top3PerArea: RDD[(String, List[(String, Int)])] = finalJoin
  .map {
    // 提取: (地区, 商品ID)
    case (productId, ((_, _, _, area), _)) =>
      ((area, productId), 1)
  }
  .reduceByKey(_ + _)
  .map {
    case ((area, productId), count) => (area, (productId, count))
  }
  .groupByKey()
  .mapValues(iter =>
    iter.toList.sortBy(_._2)(Ordering.Int.reverse).take(3))
  )

```

```

: RDD[(String, ((String, ...), ...), ...)]
: RDD[(String, String), Int]]
: RDD[(String, String), Int]]
: RDD[(String, (String, Int))]
: RDD[(String, Iterable[(String, Int)]]]
: RDD[(String, List[(String, Int)]]]

```

(6) 输出结果

最终用 `collect()` 拉回 Driver，在控制台打印每个大区及其点击量最高的 3 个商品 ID 和对应点击次数。

```

top3PerArea.collect().foreach { case (area, productList) =>
  println(s"地区: $area")
  productList.foreach { case (productId, count) =>
    println(s"\t商品ID: $productId, 点击次数: $count")
  }
}

sc.stop()
}
}

```

2.3 热门商品 Top10 统计

2.3.1 需求分析

直观展示平台中点击量最高的前 10 款商品，帮助运营快速识别最受欢迎的单品。通过 Top10 排行，能够实时察觉爆款或潜在爆款，及时调整广告、活动预算以及补货策略，同时营销、采购、物流等团队都可以基于同一榜单制定更加精准的执行方案。

2.3.2 功能实现流程

(1) 过滤与统计

首先从海量的用户行为日志中提取出“真正的点击”事件。日志中每条记录都有一个 `click_product_id` 字段，只有当它不等于 -1 时才表示一次有效点击。我们将这些点击记录映射成 `(productId, 1)`，然后调用 `reduceByKey((x, y) => x + y)` 在集群中并行地将相同 `productId` 的计数累加起来，从而得到每个商品的总体点击量。这一步充分利用了 Spark RDD 的分布式计算能力，即使面对数亿条日志也能迅速完成聚合。

(2) 名称关联

点击量本身只是一个数字，为了让结果更具可读性和业务意义，我们需要知道对应的商品名称。为此，将统计好的 `(productId, clickCount)` RDD 与事先加载好的 `product_info.txt`（映射 `productId` $\rightarrow$ `productName`）做一次 `join` 操作。这样每条记录就变成了 `(productId, (clickCount, productName))`，商品名称作为可读标签被附加到点击统计里，为后续报告和可视化提供了直观的文本信息。

(3) 排序与截取

在完成点击量统计并关联商品名称后，我们会得到一个可能包含成千上万条商品的列表。为了突出重点，只关心最受欢迎的前 10 个商品。代码先将 `(productName, clickCount)` 按点击次数降序排序（`sortBy(_._2, false)`），然后用 `take(10)` 在 Driver 端拉取前 10 条结果。这样既能保证只在本地处理小规模数据，以免网络和内存压力过大，又能够快速得到 Top10 商品列表。

(4) 可视化

有了 Top10 数据后，需要以图表形式直观呈现，便于运营和管理层快速理解。我们使用 XChart 在本地构建一个柱状图（`CategoryChart`），将商品名称作为横轴、点击次数作为纵轴。这样，读者可以一目了然地看到哪个商品最受欢迎，点击量差距有多大，也能方便地将图表嵌入到报告或仪表盘中。

2.3.3 代码实现

(1) 数据提取与点击统计

从 `user_visit_action.txt` 中读取每条用户行为，拆分字段后重点关注第 8 列的 `click_product_id`。先用 `filter(_._1 != "-1")` 去掉非点击行为，再用 `reduceByKey(_ + _)` 将所有相同 `productId` 的点击次数累加，得到一个 `(productId, clickCount)` 的 RDD。

```
val clickRDD = sc.textFile("C://Users//wei_jingxuan//Desktop//user_visit_action.txt")
  .map(line => {
    val parts = line.split("\t")
    val productId = parts(7)
    val valid = productId != "-1"
    (productId, 1)
  })
  .filter(_._1 != "-1")
  .reduceByKey(_ + _)
```

(2) 商品名称关联

读取 `product_info.txt`，生成一个 `(productId, productName)` 的映射 RDD。通过 `clickRDD.join(productRDD)` 把点击次数和商品名称对应起来，得到 `(productId, (clickCount, productName))`，于是每条数据同时拥有数值和可读名称。

```
val productRDD = sc.textFile("C://Users//wei_jingxuan//Desktop//product_info.txt")
  .map(line => {
    val parts = line.split("\t")
    val productId = parts(0)
    val productName = parts(1)
    (productId, productName)
  })

val joined = clickRDD.join(productRDD)
```

(3) Top 10 提取

将关联结果映射成 `(productName, clickCount)`，并调用 `sortBy(_._2, false)` 对点击次数降序排序。用 `take(10)` 把前 10 条最热商品拉回到 Driver 端，形成一个本地数组 `top10`。

```
val top10 = joined.map {
  case (_, (count, productName)) => (productName, count)
}
  .sortBy(_._2, ascending = false)
  .take(10)
```

(4) 可视化数据

从 `top10` 中分别抽取商品名称列表和点击次数列表，转成 Java 集合供 XChart 使用。使用 `CategoryChartBuilder` 创建一个 800×600 的空白柱状图，设置标题和坐标轴标签，同时关闭图例、旋转 X 轴标签以防止重叠，然后通过 `addSeries("点击次数", names, counts)` 加入数据。通过 `new SwingWrapper(chart).displayChart()` 在本地弹出一个 GUI 窗口，将 Top 10 热门商品的点击柱状图直观地呈现出来。

```
val chart = new CategoryChartBuilder()
  .width(800)
  .height(600)
  .title("Top10 热门商品点击数")
  .xAxisTitle("商品")
  .yAxisTitle("点击次数")
  .build()
chart.getStyler.setLegendVisible(false)
chart.getStyler.setXAxisLabelRotation(45)
chart.addSeries("点击次数", productNames, clickCounts)
val labeledNames = top10.map { case (name, count) => s"$name ($count)" }.toList.asJava
chart.updateCategorySeries("点击次数", labeledNames, clickCounts, null)
new SwingWrapper(chart).displayChart()
```

2.4 商品转化率分析

2.4.1 需求分析

明确衡量商品从“点击”到“下单”再到“支付”的转化效果，帮助运营和产品团队发现漏斗中哪一环节流失最严重。识别高潜力商品和低效商品，以便针对性地优化详情页、促销策略或支付流程，提高整体成交率。

2.4.2 功能实现流程

(1) 数据抽取与清洗

首先从原始日志中分别提取“点击”“下单”“支付”三种行为。针对点击事件，我们筛选出 `click_product_id ≠ -1` 的记录并映射成 `(productId, 1)`；针对下单和支付，由于一条日志可能对应多个商品 ID，需要对 `order_product_ids`（字段索引 9）和 `pay_product_ids`（字段索引 11）分别做 `flatMap` 拆分，再映射成 `(productId, 1)`。各自调用 `reduceByKey((x, y) => x + y)` 并行累加后，即可得到三条独立的 RDD：点击量、下单量、支付量。

(2) 全外连接合并

为了让每个商品都能同时展示点击、下单、支付数据，我们先用 `fullOuterJoin` 将“点击量”与“下单量”合并，再对结果与“支付量”再次 `fullOuterJoin`。这种全外连接确保即便某商品只出现在点击或支付过程中，也不会被丢失；在合并后通过多层 `Option`，再用 `getOrElse(0)` 将所有空缺填为 0，最终得到 `(productId, (clickCount, orderCount, payCount))`。

(3) 转化率计算

拿到每个商品的三项计数后，分别计算“下单转化率” = `orderCount / clickCount` 和“支付转化率” = `payCount / clickCount`，并格式化为百分比字符串。为了避免除以 0 的错误，若某商品的点击量为 0，则将转化率视作 0。这样就得到了一个完整的六元组 `(productId, clickCount, orderCount, payCount, orderRate, payRate)`。

以点击量为主排序，将所有商品按 `clickCount` 降序排列，调用 `collect()` 拉回 Driver，最后在控制台打印出带有商品 ID、点击数、下单数、支付数及对应转化率的表格。这样，业务团队就能一目了然地看到哪些商品不仅点击多，更有较高的下单和支付转化效率，为后续精准运营和页面优化提供了直观的数据支撑。

2.4.3 代码实现

(1) 数据加载与清洗

程序首先从日志文件中读取每行记录，并用 `split("\t")` 拆分成字段数组。由于日志中包含浏览、点击、下单、支付等多种行为，我们通过检查关键字段来筛选出感兴趣的部分：第 8 列 (`fields(7)`) 不等于 `"-1"` 的记录表示一次有效点击，第 10 列 (`fields(9)`) 不为 `"\N"` 的表示一次下单行为，第 12 列 (`fields(11)`) 不为 `"\N"` 则表示一次支付行为。这一步既剔除了无效或无关的日志行，又为后续统计提供了干净的输入。

```
val data = sc.textFile("C://Users//wei_jingxuan//Desktop//user_visit_action.txt")
    .map(_.split("\t"))
    .filter(fields => fields.length > 12)
```

(2) 多路聚合统计

点击量：对所有点击记录映射为 `(productId, 1)`，并通过 `reduceByKey(_ + _)` 得到每个商品的总点击次数。

下单量：由于一次日志可能包含多个商品 ID 的下单列表，先用 `flatMap` 拆分成多条 `(productId, 1)`，再 `reduceByKey` 聚合出每个商品的总下单次数。

支付量：处理逻辑同下单量，对支付列表字段做拆分与聚合。

```
// 点击量
val clickRDD = data
    .filter(fields => fields(7) != "-1")
    .map(fields => (fields(7), 1))
    .reduceByKey(_ + _)

// 下单量
val orderRDD = data
    .filter(fields => fields(9) != "\N")
    .flatMap(fields => fields(9).split(",").map(pid => (pid, 1)))
    .reduceByKey(_ + _)

// 支付量
val payRDD = data
    .filter(fields => fields(11) != "\N")
    .flatMap(fields => fields(11).split(",").map(pid => (pid, 1)))
    .reduceByKey(_ + _)
```

(3) 全外连接合并三路数据

使用 Spark RDD 的 `fullOuterJoin`，先将点击量与下单量做全外连接，得到每个商品的 `(Option[clickCount], Option[orderCount])` 对；再将其结果与支付量做第二次全外连接，得到 `(Option[(Option[click], Option[order])], Option[pay])`。这种全外连接保

证了无论某商品是否只出现在点击、下单或支付阶段，它都能出现在最终结果中，并将缺失的计数用 Option 表示。

```
val clickOrderJoin = clickRDD.fullOuterJoin(orderRDD)
val allJoin = clickOrderJoin.fullOuterJoin(payRDD)
```

(4) 解包与填充零值

由于连接结果中存在多层嵌套的 Option，代码通过 getOrElse 将所有 None 转成 0，最终解包成三元组 (clickCount, orderCount, payCount)。这样，每个商品都有三个明确的数字，无需担心空缺。

```
val resultRDD = allJoin.mapValues {
  case (coOpt, payOpt) =>
    val (clickOpt, orderOpt) = coOpt.getOrElse((None, None))
    val clickCount = clickOpt.getOrElse(0)
    val orderCount = orderOpt.getOrElse(0)
    val payCount = payOpt.getOrElse(0)
    (clickCount, orderCount, payCount)
}
```

(5) 转化率计算与排序输出

在得到了 (productId, (click, order, pay)) 后，分别计算“下单转化率” (order / click) 和“支付转化率” (pay / click)，并格式化为百分比字符串。最后按点击量降序对所有商品排序，调用 collect 将结果拉到 Driver，打印出包含商品 ID、点击数、下单数、支付数及对应转化率的表格。

```
val conversionRates = resultRDD.map {
  case (pid, (click, order, pay)) =>
    val orderRate = if (click == 0) 0.0 else order.toDouble / click
    val payRate = if (click == 0) 0.0 else pay.toDouble / click
    val orderRateFmt = f"${orderRate * 100}%.2f%"
    val payRateFmt = f"${payRate * 100}%.2f%"
    (pid, click, order, pay, orderRateFmt, payRateFmt)
}

.sortBy(_._2, ascending = false)
println("商品ID\t点击数\t下单数\t支付数\t下单转化率\t支付转化率")
conversionRates.collect().foreach {
  case (pid, click, order, pay, oRate, pRate) =>
    println(s"$pid\t$click\t$order\t$pay\t$oRate\t$pRate")
}
```

2.5 商品-用户关系矩阵构建

2.5.1 需求分析

在电商平台中，用户每天会产生大量行为数据，其中点击行为作为用户兴趣的第一信号，具有极高的分析价值。为了支持个性化推荐、广告精准投放、用户兴趣建模等下游任务，平台需要一种结构化方式来表达“用户对哪些商品感兴趣、

兴趣程度如何”，因此通过建立用户-商品的点击关系，构建稀疏矩阵完成这一需求。

2.5.2 功能实现流程

(1) 数据提取与清洗

代码首先读取原始日志 `user_visit_action.txt`，按制表符拆分字段，筛选出“点击行为”数据（即字段 `click_product_id` 不为 `-1` 的记录），提取出用户 ID 和商品 ID，形成 `(userId, productId)` 的对。这一步过滤掉无效行为（如浏览、下单等），聚焦于用户与商品的直接交互。

(2) 点击次数统计

将每对 `(userId, productId)` 映射为 1，再使用 `reduceByKey` 聚合，统计每位用户对每个商品的点击总数。得到的结构是 `((userId, productId), clickCount)`，为后续构建矩阵准备数据。

(3) 构建稀疏矩阵条目

为了构建稀疏矩阵，需要将用户和商品 ID 映射为连续的整数索引。代码使用 `zipWithIndex()` 给所有用户和商品分配唯一编号，并通过 `broadcast` 广播变量将映射表分发至各个计算节点，提升运行效率。借助前面生成的编码映射，将每个 `(userId, productId)` 转换为对应的 `(userIndex, productIndex)`，再将其与点击次数封装为 `MatrixEntry(rowIndex, colIndex, value)`，即稀疏矩阵的标准格式，用于后续机器学习中的推荐建模。

(4) 构建 `CoordinateMatrix` 与输出

所有条目构成一个 `CoordinateMatrix`，这是 `Spark MLlib` 提供的分布式稀疏矩阵格式，适用于大型推荐系统中用户-物品矩阵的构建。这个矩阵可以直接输入到 `ALS` 等协同过滤算法中，实现基于点击行为的个性化推荐。程序最后打印出用户数、商品数以及稀疏矩阵条目，验证矩阵是否构建正确。

2.5.3 代码实现

(1) 数据读取与点击行为过滤

这一部分从 `user_visit_action.txt` 中读取日志数据，每条记录通过字段下标提取用户 ID 和点击的商品 ID，只保留那些点击了商品的行为（即 `click_product_id != -1`），过滤掉非点击记录。

```
val raw = sc.textFile("C://Users//wei_jingxuan/Desktop//user_visit_action.txt")
  .map(_.split("\t"))
  .filter(fields => fields.length > 7 && fields(7) != "-1")
  .map(fields => (fields(1), fields(7))) // (userId, productId)
```

(2) 统计用户对每个商品的点击次数

这里统计了每个用户对每个商品的点击频次，将 (userId, productId) 作为键，对应值为点击次数，这为后续矩阵构建提供了数值基础。

```
val userProdCount: RDD[((String, String), Double)] = raw
  .map(up => (up, 1))
  .reduceByKey(_ + _)
  .map { case ((u, p), cnt) => ((u, p), cnt.toDouble) }
```

(3) 为用户和商品分配索引

先提取所有不重复的用户和商品，然后通过 zipWithIndex() 给它们分配从 0 开始的唯一编号。这一步的目的在于将用户和商品转化为矩阵中的行号和列号，便于使用 MatrixEntry(i, j, value) 构建稀疏矩阵。

```
val users: RDD[String] = userProdCount.map(_._1._1).distinct()
val prods: RDD[String] = userProdCount.map(_._1._2).distinct()

val userIndex: collection.Map[String, Long] =
  users.zipWithIndex().collectAsMap()
val prodIndex: collection.Map[String, Long] =
  prods.zipWithIndex().collectAsMap()

val bUserIndex = sc.broadcast(userIndex)
val bProdIndex = sc.broadcast(prodIndex)
```

(4) 构造矩阵元素

每一条点击统计数据都被映射为一个 MatrixEntry(row, col, value)，即：

- row 是用户编号；
- col 是商品编号；
- value 是用户对该商品的点击次数。

这一步将原始的用户-商品对转换为矩阵坐标系下的点。

```
val entries = userProdCount.map {
  case ((u, p), cnt) =>
    val ui = bUserIndex.value(u)
    val pi = bProdIndex.value(p)
    new MatrixEntry(ui, pi, cnt)
}
```

（5）构建稀疏矩阵并检查输出

将所有 `MatrixEntry` 构建成一个 `CoordinateMatrix`（坐标矩阵）。这是 Spark MLlib 中专门用于表示稀疏矩阵的格式之一，适合高维、稀疏的用户行为数据，如推荐系统中的“评分表”或“点击表”。

```
val coordMat = new CoordinateMatrix(entries)

val numUsers    = userIndex.size
val numProducts = prodIndex.size
println(s"构建完成: 用户数 = $numUsers, 商品数 = $numProducts")
println("前 10 个矩阵元素 (userIndex, productIndex, clickCount): ")
coordMat.entries.take(10).foreach(println)
```

三、运行结果展示

3.1 区域热销商品 Top3 统计

地区：华东

商品ID：86，点击次数：371

商品ID：75，点击次数：366

商品ID：47，点击次数：366

地区：西南

商品ID：1，点击次数：176

商品ID：44，点击次数：169

商品ID：60，点击次数：163

地区：西北

商品ID：15，点击次数：116

商品ID：2，点击次数：114

商品ID：22，点击次数：113

地区：华南

商品ID：23，点击次数：224

商品ID：65，点击次数：222

商品ID：50，点击次数：212

地区：东北

商品ID：41，点击次数：169

商品ID：91，点击次数：165

商品ID：93，点击次数：159

地区：华北

商品ID：99，点击次数：264

商品ID：42，点击次数：264

商品ID：19，点击次数：260

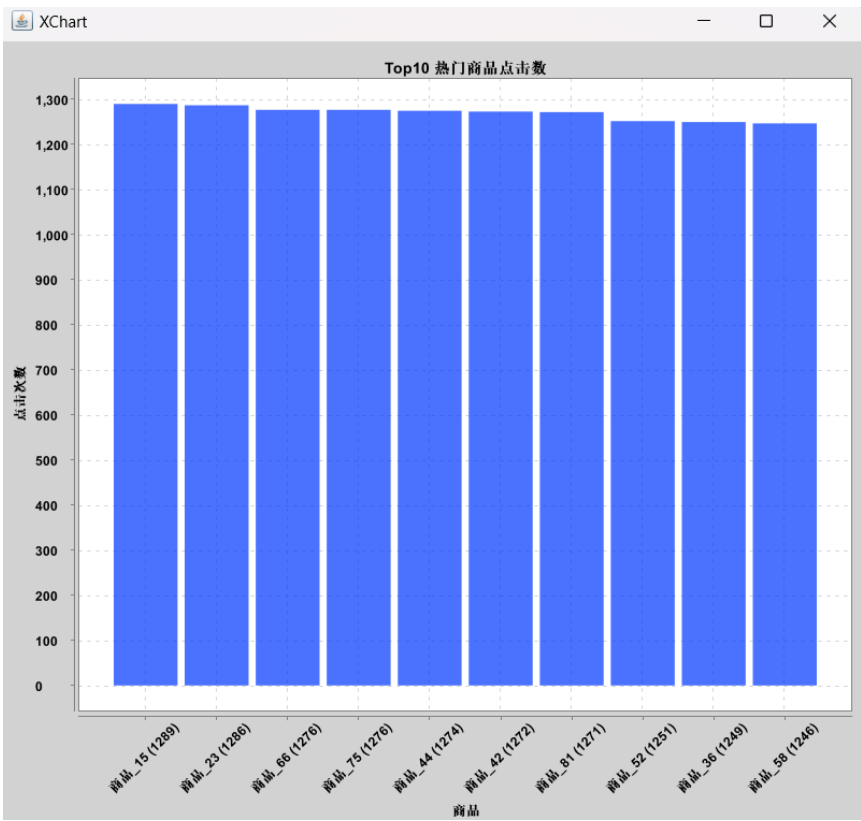
地区：华中

商品ID：62，点击次数：117

商品ID：4，点击次数：113

商品ID：29，点击次数：111

3.2 热门商品 Top10 统计



3.3 商品转化率分析

15	1289	227	162	17.61%	12.57%
23	1286	237	168	18.43%	13.06%
66	1276	206	157	16.14%	12.30%
75	1276	266	186	20.85%	14.58%
44	1274	252	164	19.78%	12.87%
42	1272	255	161	20.05%	12.66%
81	1271	265	135	20.85%	10.62%
52	1251	210	163	16.79%	13.03%
36	1249	227	180	18.17%	14.41%
58	1246	213	180	17.09%	14.45%
14	1244	230	145	18.49%	11.66%
49	1243	227	178	18.26%	14.32%
11	1242	228	148	18.36%	11.92%
55	1242	232	164	18.68%	13.20%
65	1242	240	140	19.32%	11.27%
53	1241	235	163	18.94%	13.13%
64	1239	258	172	20.82%	13.88%
37	1237	247	160	19.97%	12.93%
93	1237	245	154	19.81%	12.45%
85	1237	214	169	17.30%	13.66%

3.4 商品-用户关系矩阵构建

完整矩阵为 100*100，此处仅展示部分：

UserId	1	10	100	11	12	13	14	15	16	17
1	6	9	8	10	18	3	11	14	5	5
10	9	9	17	11	21	18	8	12	16	16
100	11	13	8	9	13	12	11	14	14	8
11	10	9	17	13	12	14	22	12	15	8
12	10	7	16	19	15	21	15	20	16	16
13	10	6	7	10	15	15	10	12	7	12
14	9	12	12	11	8	13	13	9	9	11
15	11	15	13	12	15	9	12	9	11	17
16	12	13	20	15	12	8	13	10	9	13
17	13	12	15	13	13	9	8	11	11	8

输出 CoordinateMatrix 中的 MatrixEntry 形式的前十条结果：

```
MatrixEntry(1,91,8.0)
MatrixEntry(66,13,14.0)
MatrixEntry(0,23,11.0)
MatrixEntry(55,7,12.0)
MatrixEntry(15,3,9.0)
MatrixEntry(94,60,10.0)
MatrixEntry(93,97,21.0)
MatrixEntry(49,76,10.0)
MatrixEntry(63,64,13.0)
MatrixEntry(40,4,15.0)
```

其对应解释如下：

行号	MatrixEntry	用户索引 (i)	商品索引 (j)	点击次数 (value)
1	MatrixEntry(1,91,8.0)	用户 1	商品 91	8 次点击
2	MatrixEntry(66,13,14.0)	用户 66	商品 13	14 次点击
3	MatrixEntry(0,23,11.0)	用户 0	商品 23	11 次点击
4	MatrixEntry(55,7,12.0)	用户 55	商品 7	12 次点击
5	MatrixEntry(15,3,9.0)	用户 15	商品 3	9 次点击
6	MatrixEntry(94,60,10.0)	用户 94	商品 60	10 次点击
7	MatrixEntry(93,97,21.0)	用户 93	商品 97	21 次点击
8	MatrixEntry(49,76,10.0)	用户 49	商品 76	10 次点击
9	MatrixEntry(63,64,13.0)	用户 63	商品 64	13 次点击
10	MatrixEntry(40,4,15.0)	用户 40	商品 4	15 次点击

四、总结

4.1 项目总结

本项目以电商平台的用户行为日志为数据基础，结合 Spark 分布式计算框架，完成了从数据清洗、行为提取、统计分析到可视化展示的全流程实现，系统性构建了一个高效、可扩展的电商数据分析平台。

在数据处理方面，我们借助 Spark RDD 的弹性与并行能力，成功对数百万级用户行为日志进行了清洗与转换，提取了点击、下单、支付等核心行为信息。通过多表关联（用户行为 × 城市 × 商品），不仅实现了上下文信息的丰富补全，还使得每条数据具备更强的可解释性和业务价值。

在功能模块设计上，项目实现了区域热销商品 Top3 分析、平台热门商品 Top10 排行、商品转化率评估以及用户-商品点击矩阵构建等关键任务。这些模块既覆盖了宏观市场趋势洞察，又深入到微观用户偏好建模，形成了“统计 + 推荐”的完整闭环。

在工程实现上，项目充分展示了 Spark 在实际业务中的应用潜力：利用 RDD 的 transformation/action 机制完成高效数据流转，采用全外连接与 broadcast 优化数据合并与索引映射，保障了运行效率与代码简洁性。同时借助 XChart 图形库，将分析结果转化为可交互的可视化图表，进一步提升了可读性与呈现力。

综上所述，该项目不仅锻炼了对 Spark 编程模型和大数据处理流程的掌握，也验证了数据驱动在实际电商运营中的价值。通过本项目，开发者对“数据 → 信息 → 决策”的链条有了更直观、深入的理解，也为未来深入推荐系统、实时流处理等领域打下了坚实的基础。

4.1 心得体会

通过本次基于 Spark 的电商用户行为分析项目，我不仅掌握了大数据处理的基本流程，也切身感受到了分布式计算在处理海量数据时的高效与优势。从最初的数据预处理，到行为提取、统计建模，再到可视化呈现，每一个模块的开发都让我更深入理解了 Spark RDD 的转换机制与执行逻辑。

项目中遇到的种种技术挑战，比如多表关联中的 `key` 设计、`Option` 类型的处理、数据类型转换与广播变量的使用，都促使我不断查阅文档、调试代码、优化逻辑。这种从问题出发、在实践中学习的过程，比课堂上的知识点记忆更加深刻，也让我养成了思考型编程的习惯。

此外，在实现诸如“地区热销 Top3 商品”“商品点击转化率分析”这类业务驱动的需求时，我体会到数据建模并不只是代码能力的比拼，更是对业务理解的再表达。如何用代码精确复现分析目标，如何通过合适的分组与排序实现业务逻辑的映射，是我本次收获的一大能力提升。

最后，数据可视化部分让我明白，技术服务业务的最终落脚点是“表达与洞察”。再完美的统计指标，如果不能被清晰传达给决策者，也很难发挥价值。因此我更加重视结果的展示效果与解释性，这也为我后续的分析型项目积累了宝贵经验。

总之，这次项目不仅提升了我对 `Spark` 技术栈的掌握，也锻炼了我从数据到洞察的全流程思维方式，更坚定了我未来深入大数据与智能分析方向的信心与兴趣。

附录

功能 1 完整代码

```
package com.neuedu.rdd
import org.apache.spark.{SparkConf, SparkContext}
import org.apache.spark.rdd.RDD
object Top3ProductPerArea {
  def main(args: Array[String]): Unit = {
    val conf = new
SparkConf().setAppName("Top3ProductPerArea").set
Master("local[*]")
    val sc = new SparkContext(conf)
    // 城市信息
    val cityInfoRDD: RDD[(String, (String, String))]
=
sc.textFile("C://Users//wei
jingxuan/Desktop//city_info.txt")
    .map(line => {
      val parts = line.split("\t")
      val cityId = parts(0)
      val cityName = parts(1)
      val area = parts(2)
      (cityId, (cityName, area))
    })
    // 商品信息
    val productInfoRDD: RDD[(String, String)] =
sc.textFile("C://Users//wei
jingxuan/Desktop//product_info.txt")
    .map(line => {
      val parts = line.split("\t")
      val productId = parts(0)
      val productName = parts(1)
      (productId, productName)
    })
    // 读取用户行为数据,筛选点击行为
    val clickRDD: RDD[(String, (String, String,
String))] =
sc.textFile("C://Users//wei
jingxuan/Desktop//user_visit_action.txt")
    .map(line => {
      val parts = line.split("\t")
      val date = parts(0)
      val userId = parts(1)
      val clickProductId = parts(7)
      val cityId = parts(12)

      (date, userId, clickProductId, cityId)
    })
    .filter { case (_, _, productId, _) => productId !=
"-1" }
    .map { case (date, userId, productId, cityId) =>
(cityId, (date, userId, productId))
    }
    // Join
    val clickWithCity: RDD[(String, ((String, String,
String), (String, String)))] =
clickRDD.join(cityInfoRDD)
    // 转换为 product_id 为 key, 再与 product 表
连接
    val readyForProductJoin: RDD[(String, (String,
String, String, String))] = clickWithCity.map {
      case (_, ((date, userId, productId), (cityName,
area))) =>
        (productId, (date, userId, cityName, area))
    }
    val finalJoin: RDD[(String, ((String, String,
String, String), String))] =
readyForProductJoin.join(productInfoRDD)
    // 6. 统计每个地区每个商品的点击次数
    val top3PerArea: RDD[(String, List[(String, Int)]]
= finalJoin
    .map {
      case (productId, ((_, _, _, area), _)) =>
        ((area, productId), 1)
    }
    .reduceByKey(_ + _)
    .map {
      case ((area, productId), count) => (area,
(productId, count))
    }
    .groupByKey()
    .mapValues(iter =>
iter.toList.sortBy(_._2)(Ordering.Int.reverse).take(3)
    )
    // 输出
    top3PerArea.collect().foreach { case (area,
```

```

productList) =>
    println(s"地区: $area")
    productList.foreach { case (productId, count) =>
        println(s"\t 商品 ID: $productId, 点击次数:
$count")
    }
}
sc.stop()
}
}

```

功能 2 完整代码

```

package com.neuedu.rdd

import org.apache.spark.{SparkConf, SparkContext}
import org.knowm.xchart.{CategoryChart,
CategoryChartBuilder, SwingWrapper}
import org.knowm.xchart.style.Styler
import scala.jdk.CollectionConverters._

object ProductClickTop10Chart {
    def main(args: Array[String]): Unit = {
        val conf = new
SparkConf().setAppName("ProductClickTop10Chart")
.setMaster("local[*]")

        val sc = new SparkContext(conf)

        // 读取,过滤点击行为
        val clickRDD = sc.textFile("C://Users//wei
jingxuan//Desktop//user_visit_action.txt")

        .map(line => {
            val parts = line.split("\t")
            val productId = parts(7)
            val valid = productId != "-1"
            (productId, 1)
        })
        .filter(_._1 != "-1")
        .reduceByKey(_ + _)

        // 获取商品名称
        val productRDD = sc.textFile("C://Users//wei
jingxuan//Desktop//product_info.txt")
        .map(line => {

```

```

            val parts = line.split("\t")
            val productId = parts(0)
            val productName = parts(1)
            (productId, productName)
        })

        val joined = clickRDD.join(productRDD)

        // 转换为 (productName, count), 取前 10
        val top10 = joined.map {
            case (_, (count, productName)) =>
                (productName, count)
        }
        .sortBy(_._2, ascending = false)
        .take(10)

        // 准备绘图数据
        val productNames =
top10.map(_._1).toList.asJava
        val clickCounts =
top10.map(_._2.asInstanceOf[Number]).toList.asJava

        // 构建图表
        val chart = new CategoryChartBuilder()
            .width(800)
            .height(600)
            .title("Top10 热门商品点击数")
            .xAxisTitle("商品")
            .yAxisTitle("点击次数")
            .build()
        chart.getStyler.setLegendVisible(false)
        chart.getStyler.setXAxisLabelRotation(45)
        chart.addSeries(" 点 击 次 数 ", productNames,
clickCounts)

        val labeledNames = top10.map { case (name,
count) => s"$name ($count)" }.toList.asJava
        chart.updateCategorySeries(" 点 击 次 数 ",
labeledNames, clickCounts, null)
        new SwingWrapper(chart).displayChart()

        sc.stop()
    }
}

```

功能 3 完整代码

```
package com.neuedu

import org.apache.spark.{SparkConf, SparkContext}

object ProductConversionAnalysis {
  def main(args: Array[String]): Unit = {

    val conf = new SparkConf()
      .setAppName("ProductConversionAnalysis")
      .setMaster("local[*]")
    val sc = new SparkContext(conf)

    val data = sc.textFile("C://Users//weijingxuan//Desktop//user_visit_action.txt")
      .map(_._split("\t"))
      .filter(fields => fields.length > 12)

    // 点击量
    val clickRDD = data
      .filter(fields => fields(7) != "-1")
      .map(fields => (fields(7), 1))
      .reduceByKey(_ + _)

    // 下单量
    val orderRDD = data
      .filter(fields => fields(9) != "\\N")
      .flatMap(fields => fields(9).split(",").map(pid => (pid, 1)))
      .reduceByKey(_ + _)

    // 支付量
    val payRDD = data
      .filter(fields => fields(11) != "\\N")
      .flatMap(fields => fields(11).split(",").map(pid => (pid, 1)))
      .reduceByKey(_ + _)

    // 全外连接
    val clickOrderJoin = clickRDD.fullOuterJoin(orderRDD)
    val allJoin = clickOrderJoin.fullOuterJoin(payRDD)

    // 解包所有 Option, 得到 (pid, (click, order, pay))
```

```
    val resultRDD = allJoin.mapValues {
      case (coOpt, payOpt) =>
        val (clickOpt, orderOpt) = coOpt.getOrElse((None, None))
        val clickCount = clickOpt.getOrElse(0)
        val orderCount = orderOpt.getOrElse(0)
        val payCount = payOpt.getOrElse(0)
        (clickCount, orderCount, payCount)
    }

    // 计算转化率并排序
    val conversionRates = resultRDD.map {
      case (pid, (click, order, pay)) =>
        val orderRate = if (click == 0) 0.0 else
          order.toDouble / click
        val payRate = if (click == 0) 0.0 else
          pay.toDouble / click
        val orderRateFmt = f"${orderRate * 100}%0.2f%%"
        val payRateFmt = f"${payRate * 100}%0.2f%%"
        (pid, click, order, pay, orderRateFmt, payRateFmt)
    }
    .sortBy(_._2, ascending = false)
    println("商品 ID\t 点击数\t 下单数\t 支付数\t 下单转化率\t 支付转化率")
    conversionRates.collect().foreach {
      case (pid, click, order, pay, oRate, pRate) =>
        println(s"$pid\t$click\t$order\t$pay\t$oRate\t$pRate")
    }

    sc.stop()
  }
}
```

功能 4 完整代码

```
package com.neuedu

import org.apache.spark.{SparkConf, SparkContext}
import org.apache.spark.rdd.RDD
import org.apache.spark.mllib.linalg.distributed.{Coordinate
```

```

Matrix, MatrixEntry}

object UserProductClickMatrix {
  def main(args: Array[String]): Unit = {
    val conf = new SparkConf()
      .setAppName("UserProductClickMatrix")
      .setMaster("local[*]")
    val sc = new SparkContext(conf)

    // 读取用户行为，过滤出点击
    (click_product_id != "-1")
    val raw = sc.textFile("C://Users//weijingxuan//Desktop//user_visit_action.txt")
      .map(_ .split("\t"))
      .filter(fields => fields.length > 7 && fields(7) != "-1")
      .map(fields => (fields(1), fields(7))) // (userId, productId)

    // 统计每个 (userId, productId) 的点击次数
    val userProdCount: RDD[(String, String), Double] = raw
      .map(up => (up, 1))
      .reduceByKey(_ + _)
      .map { case ((u, p), cnt) => ((u, p), cnt.toDouble) }

    // 分配连续索引
    val users: RDD[String] = userProdCount.map(_._1._1).distinct()
    val prods: RDD[String] = userProdCount.map(_._1._2).distinct()

    val userIndex: collection.Map[String, Long] = users.zipWithIndex().collectAsMap()
    val prodIndex: collection.Map[String, Long] = prods.zipWithIndex().collectAsMap()

    val bUserIndex = sc.broadcast(userIndex)
    val bProdIndex = sc.broadcast(prodIndex)

    // 构建 MatrixEntry
    val entries = userProdCount.map {
      case ((u, p), cnt) =>
        val ui = bUserIndex.value(u)
        val pi = bProdIndex.value(p)
        new MatrixEntry(ui, pi, cnt)
    }

    // output
    val coordMat = new CoordinateMatrix(entries)

    val numUsers = userIndex.size
    val numProducts = prodIndex.size
    println(s"构建完成：用户数 = $numUsers，商品数 = $numProducts")
    println("前 10 个矩阵元素 (userIndex, productIndex, clickCount): ")
    coordMat.entries.take(10).foreach(println)

    sc.stop()
  }
}

```