Spark MLlib 推荐系统快速入门

课程说明

- 了解什么是Spark MLlib
- 了解Spark MLlib所支持的数据类型
- 掌握RDD、DataSet、Dataframe区别及转化
- 了解什么是推荐系统
- 了解协同过滤算法
- 了解相似度算法
- 掌握交替最小二乘法
- 掌握使用交替最小二乘法实现推荐系统

1、Spark MLlib

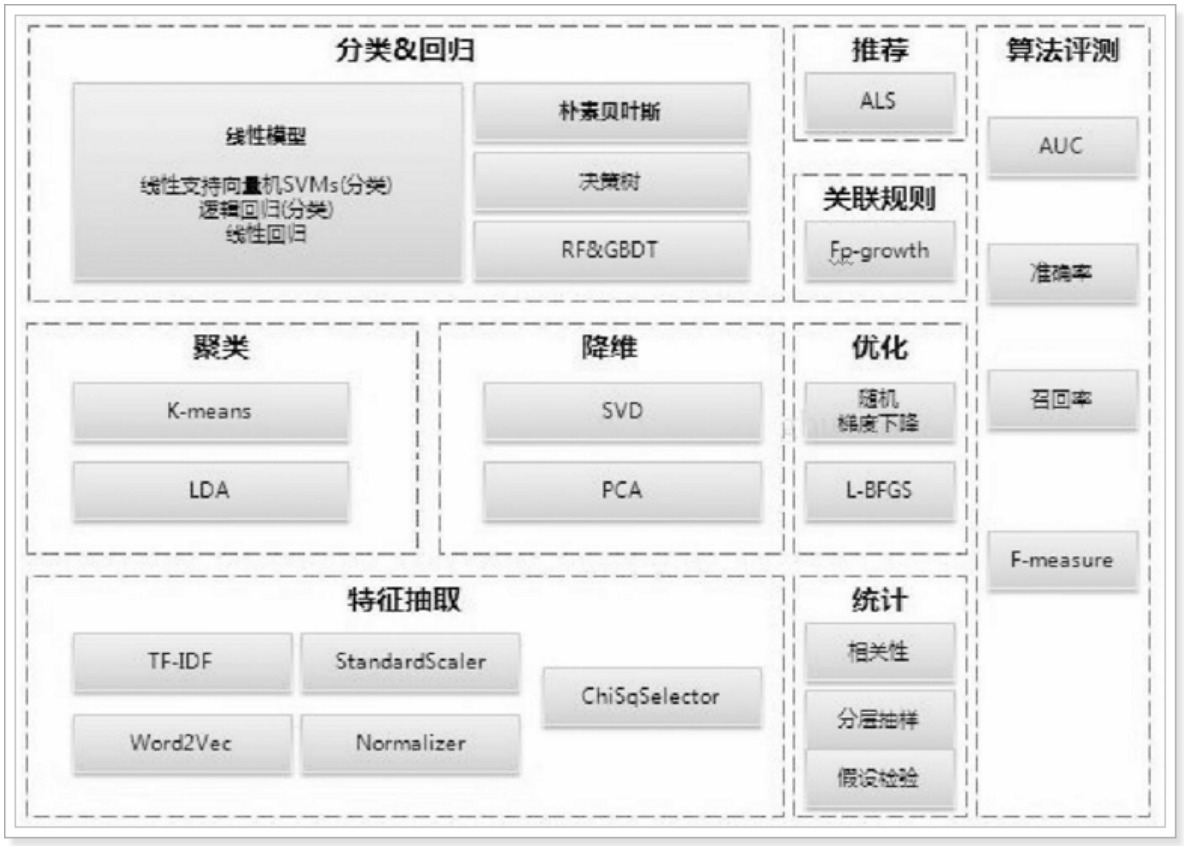
1.1、什么是Spark MLlib ?

MLlib是Spark的机器学习（ML）库。旨在简化机器学习的工程实践工作，并方便扩展到更大规模。MLlib由一些通用的学习算法和工具组成，包括分类、回归、聚类、协同过滤、降维等，同时还包括底层的优化原语和高层的管道API。

MLlib目前分为两个代码包：

- spark.mllib 包含基于RDD的原始算法API。
- spark.ml 则提供了基于DataFrames 高层次的API，可以用来构建机器学习管道。

MLlib支持的算法库如下：



1.2、支持的数据类型

MLlib支持的数据类型是比较丰富的，从最基本的Spark数据集RDD到部署在集群中向量和矩阵，并且还支持部署在本地计算机中的本地化格式。

类型名称	释 义
Local vector	本地向量集。主要向 Spark 提供一组可进行操作的数据集合
Labeled point	向量标签。让用户能够分类不同的数据集合
Local matrix	本地矩阵。将数据集合以矩阵形式存储在本地计算机中
Distributed matrix	分布式矩阵。将数据集合以矩阵形式存储在分布式计算机中

1.2.1、本地向量集

MLlib使用的本地化存储类型是向量，这里的向量主要由两类构成：稀疏型数据集（sparse）和密集型数据集（dense）。

导入MLlib的依赖：

```
<dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-mllib_2.12</artifactId>
    <version>2.4.3</version>
</dependency>
```

1.2.1.1、密集型数据集

```
package cn.itcast.spark;

import org.apache.spark.mllib.linalg.Vector;
import org.apache.spark.mllib.linalg.Vectors;
import org.junit.Test;

public class SparkMLlib {

    @Test
    public void testDense(){
        Vector vd = Vectors.dense(9, 5, 2, 7); //定义密集型向量
        double v = vd.apply(2); //获取下标为2的值
        System.out.println(v); //2
    }
}

```

1.2.1.2、稀疏型数据集

```
    @Test
    public void testSparse() {
        int[] indexes = new int[]{0, 1, 2, 5};
        double[] values = new double[]{9, 5, 2, 7};
        Vector vd = Vectors.sparse(6, indexes, values); //定义稀疏型向量
        double v = vd.apply(2); //获取下标为2的值
        double v2 = vd.apply(4); //获取下标为4的值
        double v3 = vd.apply(5); //获取下标为5的值
        System.out.println(v); //2
        System.out.println(v2); //0
        System.out.println(v3); //7
    }
```

1.2.2、向量标签

向量标签用于MLLIB中机器学习算法做标记。在分类问题中，可以将不同的数据集分成若干份，以整数型0、1、2进行标记。例如：垃圾邮件标记为1，非垃圾邮件标记为0。

```
    @Test
    public void testLabeledPoint(){
        Vector vd = Vectors.dense(9, 5, 2, 7); //定义密集型向量
        LabeledPoint lp = new LabeledPoint(1, vd); //定义标签向量
        System.out.println(lp.features());
        System.out.println(lp.label());
        System.out.println(lp);
        System.out.println("------------");

        int[] indexes = new int[]{0, 1, 2, 3};
        double[] values = new double[]{9, 5, 2, 7};
        Vector vd2 = Vectors.sparse(4, indexes, values); //定义稀疏型向量
        LabeledPoint lp2 = new LabeledPoint(2, vd2);
        System.out.println(lp2.features());
        System.out.println(lp2.label());
        System.out.println(lp2);
    }
```

1.2.3、本地矩阵

大数据运算中，为了更好地提升计算效率，可以更多地使用矩阵运算进行数据处理。部署在单机中的本地矩阵就是一个很好的存储方法。举一个简单的例子，例如一个数组Array（1,2,3,4,5,6），将其分为2行3列的：

```
    @Test
    public void testMatrix() {
        double[] values = new double[]{1, 2, 3, 4, 5, 6};
        Matrix mx = Matrices.dense(2, 3, values);
        System.out.println(mx);
    }
```

运行结果：

```
1.0  3.0  5.0
2.0  4.0  6.0
```

1.2.4、分布式矩阵

分布式矩阵由长整型行列索引和双精度浮点型值数据组成，分布式存储在一个或多个RDD中，对于巨大的分布式矩阵来说，选择正确的存储格式非常重要，将一个分布式矩阵转化为另外一个不同格式需要混洗(shuffle)，其代价很高。在MLlib实现了四类分布式矩阵存储格式，分别是：

- 行矩阵（RowMatrix）
- 行索引矩阵（IndexedRowMatrix）
- 坐标矩阵（CoordinateMatrix）
- 分块矩阵（BlockMatrix）

1.2.4.1、行矩阵

行矩阵是最基本的一种矩阵类型。行矩阵是以行作为基本方向的矩阵存储格式，列的作用相对较小。可以将其理解为行矩阵是一个巨大的特征向量的集合。每一行就是一个具有相同格式的向量数据，且每一行的向量内容都可以单独取出来进行操作。

```
a.txt:
1 2 3
4 5 6

@Test
    public void testRowMatrix() {
        SparkConf sparkConf = new SparkConf()
                .setAppName("SparkMLlib")
                .setMaster("local[*]");
        JavaSparkContext jsc = new JavaSparkContext(sparkConf);

        JavaRDD<String> rdd1 = jsc.textFile("F:\\code\\files\\a.txt");

        JavaRDD<Vector> rdd2 = rdd1.map(v1 -> {
            String[] ss = v1.split(" ");
            double[] ds = new double[ss.length];
            for (int i = 0; i < ss.length; i++) {
                ds[i] = Double.valueOf(ss[i]);
            }
            return Vectors.dense(ds);
        });

        RowMatrix rmx = new RowMatrix(rdd2.rdd());

        System.out.println(rmx.numRows()); //2
        System.out.println(rmx.numCols()); //3
    }
```

1.2.4.2、行索引矩阵

An IndexedRowMatrix类似于a RowMatrix但具有有意义的行索引。它由索引行的RDD支持，因此每行由其索引（long-typed）和本地向量表示。

```
@Test
    public void testIndexedRowMatrix() {
        SparkConf sparkConf = new SparkConf()
                .setAppName("SparkMLlib")
                .setMaster("local[*]");
        JavaSparkContext jsc = new JavaSparkContext(sparkConf);

```

```
        JavaRDD<String> rdd1 = jsc.textFile("F:\\code\\files\\a.txt");

        JavaRDD<Vector> rdd2 = rdd1.map(v1 -> {
            String[] ss = v1.split(" ");
            double[] ds = new double[ss.length];
            for (int i = 0; i < ss.length; i++) {
                ds[i] = Double.valueOf(ss[i]);
            }
            return Vectors.dense(ds);
        });

        JavaRDD<IndexedRow> rdd3 = rdd2.map(v1 -> new IndexedRow(v1.size(),
v1)); //建立待索引的行

        IndexedRowMatrix irmx = new IndexedRowMatrix(rdd3.rdd()); //行索引


        JavaRDD<IndexedRow> rdd4 = irmx.rows().toJavaRDD();
        List<IndexedRow> collect = rdd4.collect();
        collect.forEach(indexedRow -> System.out.println(indexedRow)); //打
印行内容
    }
```

1.2.4.3、坐标矩阵

CoordinateMatrix是由其条目的RDD支持的分布式矩阵。每个条目都是一个元组(i: Long, j: Long, value: Double)，其中i是行索引，j是列索引， value是条目值。只有当矩阵的两个维度都很大并且矩阵非常稀疏时，才应该使用它。

```
@Test
    public void testCoordinateMatrix() {
        SparkConf sparkConf = new SparkConf()
                .setAppName("SparkMLlib")
                .setMaster("local[*]");
        JavaSparkContext jsc = new JavaSparkContext(sparkConf);

        JavaRDD<String> rdd1 = jsc.textFile("F:\\code\\files\\a.txt");

        JavaRDD<MatrixEntry> rdd2 = rdd1.map(v1 -> {
            String[] ss = v1.split(" ");
            return new
MatrixEntry(Long.valueOf(ss[0]),Long.valueOf(ss[1]),Double.valueOf(ss[2]));
        });

        CoordinateMatrix matrix = new CoordinateMatrix(rdd2.rdd());

        List<MatrixEntry> collect = matrix.entries().toJavaRDD().collect();
        for (MatrixEntry matrixEntry : collect) {
            System.out.println(matrixEntry);
        }

    }
```

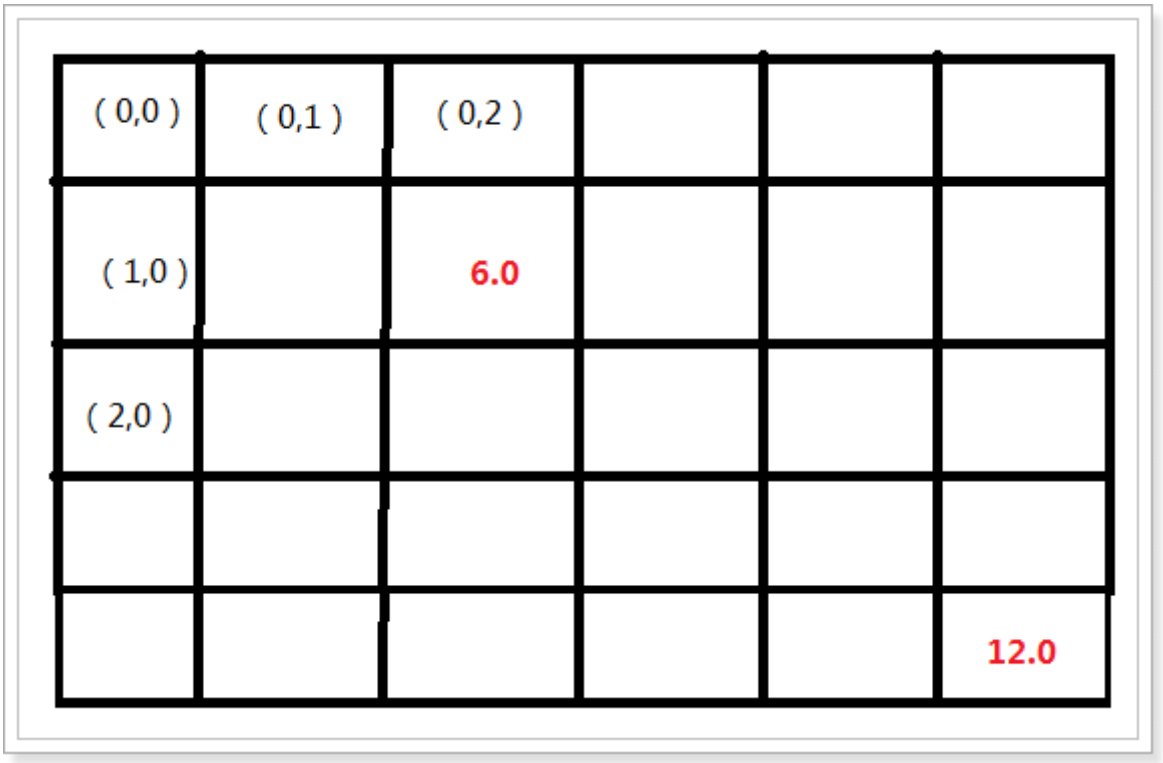
1.2.4.4、分块矩阵

BlockMatrix是支持矩阵分块RDD的分布式矩阵，其中矩阵分块由((int,int),matrix)元祖所构成（int,int）是该部分矩阵所处的矩阵的索引位置，Matrix表示该索引位置上的子矩阵

分块矩阵支持矩阵加法和乘法，并设有辅助函数验证用于检查矩阵是否设置正确。

```
@Test
    public void testCoordinateMatrix() {
        SparkConf sparkConf = new SparkConf()
                .setAppName("SparkMLlib")
                .setMaster("local[*]");
        JavaSparkContext jsc = new JavaSparkContext(sparkConf);

        JavaRDD<String> rdd1 = jsc.textFile("F:\\code\\files\\a.txt");

        JavaRDD<MatrixEntry> rdd2 = rdd1.map(v1 -> {
            String[] ss = v1.split(" ");
            return new
MatrixEntry(Long.valueOf(ss[0]),Long.valueOf(ss[1]),Double.valueOf(ss[2]));
        });

        CoordinateMatrix matrix = new CoordinateMatrix(rdd2.rdd());

        //转化成块矩阵
        BlockMatrix blockMatrix = matrix.toBlockMatrix();

        // 对该分块矩阵进行检验，确认该分块是否正确，如果不正确则抛出异常
        blockMatrix.validate();

        BlockMatrix addBlockMatrix = blockMatrix.add(blockMatrix); // 相加

        List<Tuple2<Tuple2<Object, Object>, Matrix>> collect =
addBlockMatrix.blocks().toJavaRDD().collect();
        for (Tuple2<Tuple2<Object, Object>, Matrix> tuple2MatrixTuple2 :
collect) {
            System.out.println(tuple2MatrixTuple2);
        }

//        ((0,0),5 x 6 CSCMatrix
//                (1,2) 6.0
//        (4,5) 12.0)

    }
```

结果说明：



1.3、RDD、DataSet、Dataframe区别及转化

RDD(Spark1.0)--->DataFrame(Spark1.3)---->DataSet(Spark1.6)

SparkSql提供了Dataframe和DataSet的数据抽象，DataFrame就是RDD+Schema，可以认为是一张二维表格。它的劣势是在编译器不对表格中的字段进行类型检查。在运行期间检查。DataSet是Spark最新数据抽象，Spark的发展会逐步将DataSet作为主要的数据抽象，弱化RDD和DataFrame。DataSet包含了DataFrame所有的优化机制。除此之外提供了以样例类为Schema模型的强类型。

示例：

```
package cn.itcast.spark;

import java.io.Serializable;

public class Student implements Serializable {

   private Long id;
   private String name;
   private Integer age;

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
```

```
    }

    public Integer getAge() {
        return age;
    }

    public void setAge(Integer age) {
        this.age = age;
    }

    @Override
    public String toString() {
        return "Student{" +
                "id=" + id +
                ", name='" + name + '\'' +
                ", age=" + age +
                '}';
    }
}
```

```
users.txt:
1,zhangsan,20
2,lisi,21
3,wangwu,19
4,zhaoliu,18
```

```
 @Test
    public void testDataSet() {
        SparkSession sparkSession = SparkSession.builder().appName("SparkMLlib").master("local[*]").getOrCreate();

        // 读取文件
        JavaRDD<String> source = sparkSession.read().textFile("F:\\code\\users.txt").toJavaRDD();

        JavaRDD<Student> rowRDD = source.map(line -> {
            String parts[] = line.split(",");

            Student student = new Student();
            student.setId(Long.valueOf(parts[0]));
            student.setName(parts[1]);
            student.setAge(Integer.valueOf(parts[2]));
            return student;
        });

        Dataset<Row> df = sparkSession.createDataFrame(rowRDD, Student.class);
//        df.show();
        df.select("id", "name").orderBy(df.col("id").desc()).show();

    }
```

运行结果：

```
+---+--------+
| id|    name|
+---+--------+
|  4| zhaoliu|
|  3|  wangwu|
|  2|    lisi|
|  1|zhangsan|
+---+--------+
```

2、推荐系统

2.1、从广告说起

先如今，广告可谓是无处不在，报纸、电视、视频网站、短信、邮件等等。

弹框广告：

> 未来广告：精准推荐，不再让人们对广告反感，而是会感觉到惊讶。只要做到精准，“广告”就不再是“广告”。

2.2、什么是推荐系统？

为了解决信息过载和用户无明确需求的问题，找到用户感兴趣的物品，才有了个性化推荐系统。

其实，解决信息过载的问题，代表性的解决方案是分类目录和搜索引擎，如hao123，电商首页的分类目录以及百度，360搜索等。

不过分类目录和搜索引擎只能解决用户主动查找信息的需求，即用户知道自己想要什么，并不能解决用户没用明确需求很随便的问题。经典语录是：你想吃什么，随便！面对这种很随便又得罪不起的用户（女友和上帝），只能**通过分析用户的历史行为给用户的兴趣建模**，从而主动给用户推荐能够满足他们兴趣和需求的信息。比如问问女友的闺蜜，她一般什么时候喜欢吃什么。

2.3、电商是推荐系统的先行者

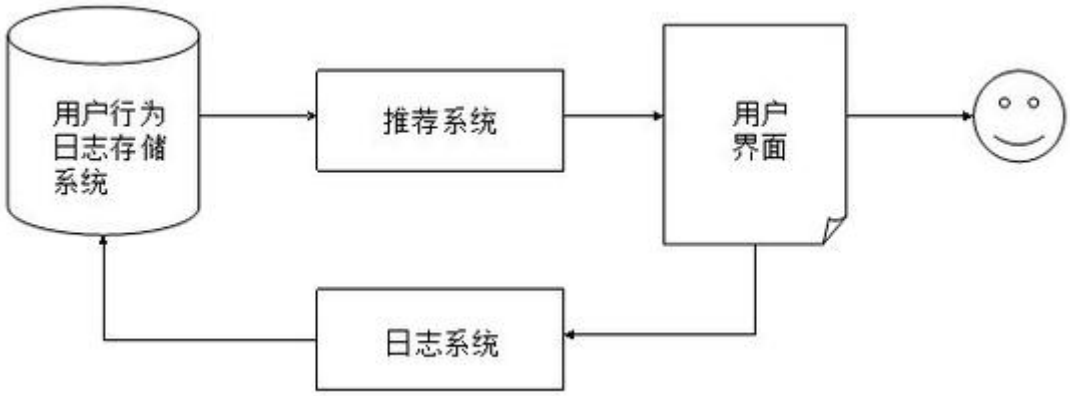
- [ ] 电子商务网站是个性化推荐系统重要地应用的领域之一，亚马逊就是个性化推荐系统的积极应用者和推广者，亚马逊的推荐系统深入到网站的各类商品，为亚马逊带来了至少30%的销售额。
- [ ] 不光是电商类，推荐系统无处不在。QQ，微信的好友推荐；新浪微博的你可能感兴趣的人；优酷，土豆的电影推荐；豆瓣的图书推荐；大从点评的餐饮推荐；脉脉的同事推荐等。
- [ ] 推荐引擎的鼻祖思想源泉：http://portal.acm.org/citation.cfm?id=1070751
- [ ] 亚马逊最早提出基于物品的协同过滤推荐算法：http://portal.acm.org/citation.cfm?id=372071

京东的推荐：

淘宝的推荐：

当当网的推荐：

2.4、推荐系统业务流程



推荐系统广泛存在于各类网站中，作为一个应用为用户提供个性化的推荐。它需要一些用户的历史数据，一般由三个部分组成：基础数据、推荐算法系统、前台展示。

- 基础数据包括很多维度，包括用户的访问、浏览、下单、收藏，用户的历史订单信息，评价信息等很多信息；
- 推荐算法系统主要是根据不同的推荐诉求由多个算法组成的推荐模型；
- 前台展示主要是对客户端系统进行响应，返回相关的推荐信息以供展示。

2.5、推荐系统所涉及到的知识

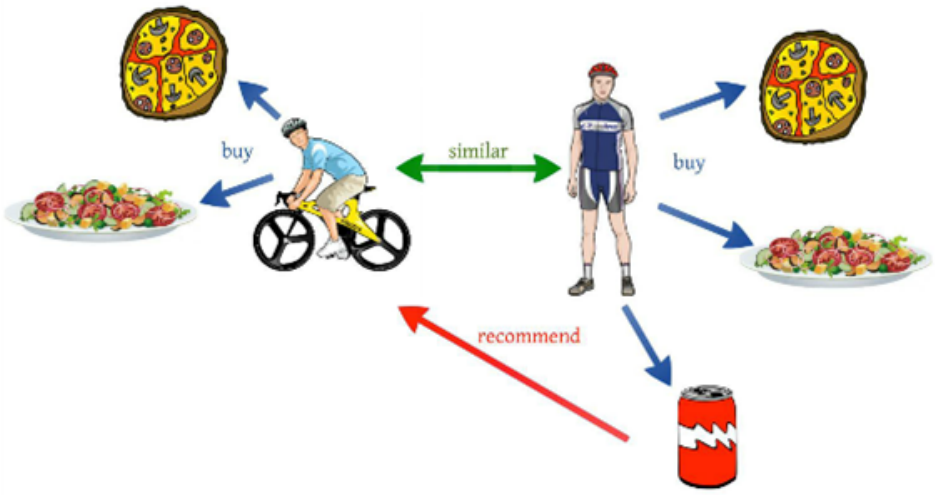
- [ ] 电子商务、交友等业务知识
- [ ] 网站架构和运营
- [ ] 机器学习算法，数学建模
- [ ] 大数据计算平台

3、协同过滤算法

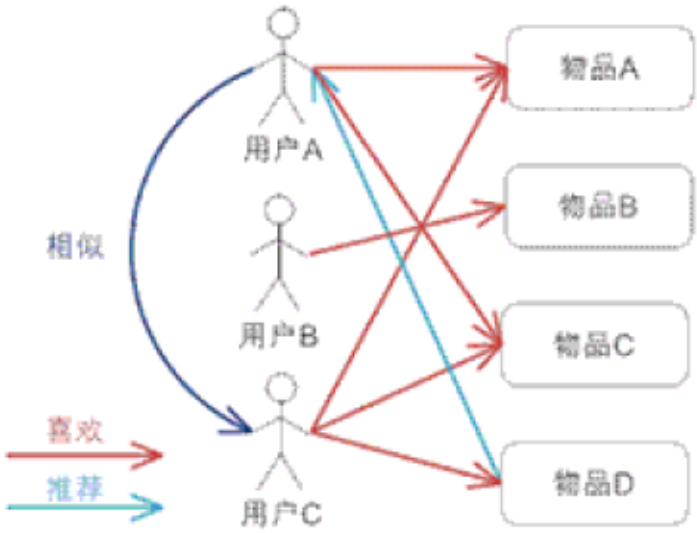
迄今为止，在个性化推荐系统中，协同过滤技术是应用最成功的技术。目前国内外有许多大型网站应用这项技术为用户更加智能（个性化、千人千面）的推荐内容。

> 核心思想：
>
> 协同过滤一般是在海量的用户中发掘出一小部分和你品位比较类似的，在协同过滤中，这些用户成为邻居，然后根据他们喜欢的其他东西组织成一个排序的目录作为推荐给你。
>
> 问题：
>
> 如何确定一个用户是和你有相似的品位？ 如何将邻居们的喜好组织成一个排序的目录？

3.1、基于用户的推荐 UserCF



用户/物品	物品A	物品B	物品C	物品D
用户A	√		√	推荐
用户B		√		
用户C	√		√	√

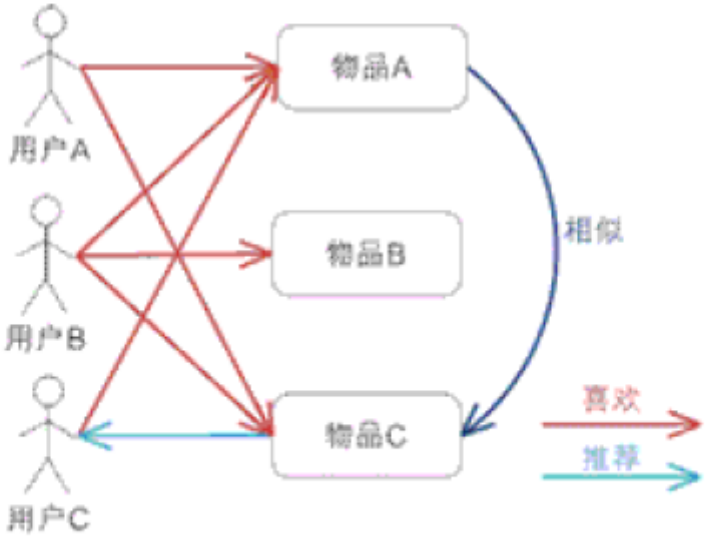


对于用户A，根据用户的历史偏好，这里只计算得到一个邻居–用户C，然后将用户C 喜欢的物品D 推荐给用户A。

基于用户的协同过滤算法先计算的是用户与用户的相似度（兴趣相投，物以类聚人以群分），然后将相似度比较接近的用户A购买的物品推荐给用户B，专业的说法是该算法用最近邻居（nearest-neighbor）算法找出一个用户的邻居集合，该集合的用户和该用户有相似的喜好，算法根据邻居的偏好对该用户进行预测。

3.2、基于商品的推荐 ItemCF

用户/物品	物品A	物品B	物品C
用户A	√		√
用户B	√	√	√
用户C	√		推荐



- 基于ItemCF的原理和基于UserCF类似，只是在计算邻居时采用物品本身，而不是从用户的角度，即基于用户对物品的偏好找到相似的物品，然后根据用户的历史偏好，推荐相似的物品给他。
- 从计算的角度看，就是将所有用户对某个物品的偏好作为一个向量来计算物品之间的相似度，得到物品的相似物品后，根据用户历史的偏好预测当前用户还没有表示偏好的物品，计算得到一个排序的物品列表作为推荐。
- 解释：对于物品A，根据所有用户的历史偏好，喜欢物品A 的用户都喜欢物品C，得出物品A 和物品C 比较相似，而用户C 喜欢物品A，那么可以推断出用户C 可能也喜欢物品C。

3.3、如何选择？

- 在社交项目中，如微信、QQ，显然选择基于用户推荐比较好，因为推荐往往都是和人相关的。
  - 如：在QQ登录后，会有提示，好友的好友可能是你认识的，推荐给你添加好友。
- 在电商项目中，用户的数量远大于商品数量，所以基于商品的推荐的复杂度要低，而且也比较合理。
- 其实，在实际的推荐系统中，往往不单是使用一种推荐，而是会多种推荐混合使用。
- 所以，选择基于用户还是基于商品的推荐，和应用场景有很大的关系。

4、用户偏好收集

用户和商品的之间的关系，什么样的数据能够表示用户对商品的喜好呢？ 要从用户的行为和偏好中发现规律，并基于此给予推荐，如何收集用户的偏好信息成为系统推荐效果最基础的决定因素。用户有很多方式向系统提供自己的偏好信息，而且不同的应用也可能大不相同，下面举例进行介绍：

表 1 用户行为和用户偏好

用户行为	类型	特征	作用
评分	显式	整数量化的偏好，可能的取值是 [0, n]；n 一般取值为5 或者是 10	通过用户对物品的评分，可以精确的得到用户的偏好
投票	显式	布尔量化的偏好，取值是 0 或 1	通过用户对物品的投票，可以较精确的得到用户的偏好
转发	显式	布尔量化的偏好，取值是 0 或 1	通过用户对物品的投票，可以精确的得到用户的偏好。 如果是站内，同时可以推理得到被转发人的偏好（不精确）
保存书签	显式	布尔量化的偏好，取值是 0 或 1	通过用户对物品的投票，可以精确的得到用户的偏好。
标记标签(Tag)	显式	一些单词，需要对单词进行分析，得到偏好	通过分析用户的标签，可以得到用户对项目的理解，同时可以分析出用户的情感：喜欢还是讨厌
评论	显式	一段文字，需要进行文本分析，得到偏好	通过分析用户的评论，可以得到用户的情感：喜欢还是讨厌
点击流(查看)	隐式	一组用户的点击，用户对物品感兴趣，需要进行分析，得到偏好	用户的点击一定程度上反映了用户的注意力，所以它也可以从一定程度上反映用户的喜好。
页面停留时间	隐式	一组时间信息，噪音大，需要进行去噪，分析，得到偏好	用户的页面停留时间一定程度上反映了用户的注意力和喜好，但噪音偏大，不好利用。
购买	隐式	布尔量化的偏好，取值是 0 或 1	用户的购买是很明确的说明这个项目它感兴趣。

4.1、数据的降噪和归一化

收集了用户行为数据，我们还需要对数据进行一定的预处理，其中最核心的工作就是：降噪和归一化。

- 降噪：用户行为数据是用户在使用应用过程中产生的，它可能存在大量的噪音和用户的误操作，我们可以通过经典的数据挖掘算法过滤掉行为数据中的噪音，这样可以是我们的分析更加精确。
- 归一化：如前面讲到的，在计算用户对物品的喜好程度时，可能需要对不同的行为数据进行加权。但可以想象，不同行为的数据取值可能相差很 大，比如，用户的查看数据必然比购买数据大的多，如何将各个行为的数据统一在一个相同的取值范围中，从而使得加权求和得到的总体喜好更加精确，就需要我们 进行归一化处理。最简单的归一化处理，就是将各类数据除以此类中的最大值，以保证归一化后的数据取值在 [0,1] 范围中。

5、相似度算法

无论是基于用户还是基于商品的推荐，都是需要找到相似的用户或者商品，才能做推荐，所以，相似度算法就变得非常重要了。

常见的相似度算法有：

- 欧几里德距离算法（Euclidean Distance）
- 皮尔逊相似度算法（Pearson Correlation Coefficient）
- 基于夹角余弦相似度算法（Consine Similarity）
- 基于Tanimoto系数相似度（Tanimoto Coefficient）

5.1、欧几里德距离算法

计算公式

编辑

二维空间的公式

$\rho = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$,$|X| = \sqrt{x_2^2 + y_2^2}$. 其中， ρ 为点 (x_2, y_2) 与点 (x_1, y_1) 之间的欧氏距离； $|X|$ 为点 (x_2, y_2) 到原点的欧氏距离。

三维空间的公式

$$\rho = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2},$$

$$|X| = \sqrt{x_2^2 + y_2^2 + z_2^2}.$$

n维空间的公式

$$d(x, y) := \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + \cdots + (x_n - y_n)^2} = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}.$$

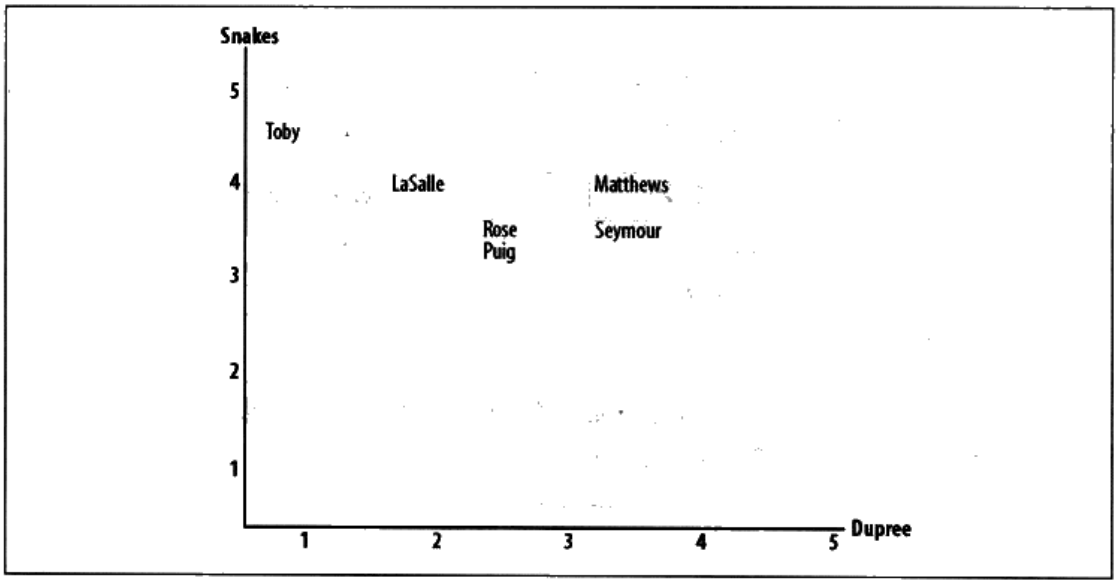


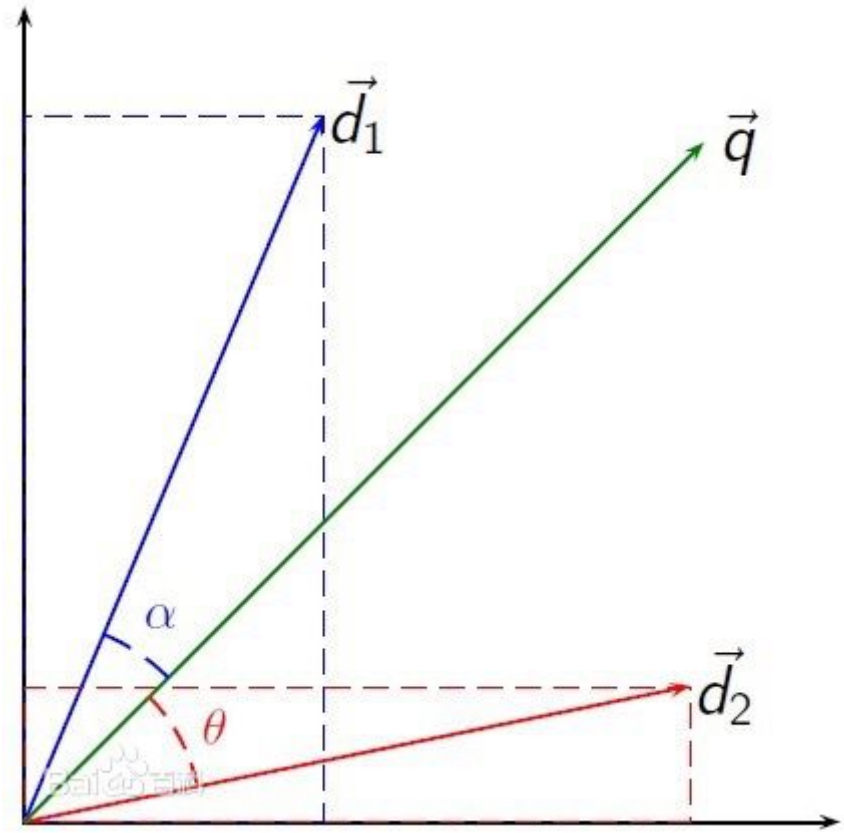
图 2-1：处于“偏好空间”中的人们

上图即二维空间中6位用户对Snakes 和 Dupree 这两Item评价的直观体现。

根据两用户之间共同评价的Item为维度，建立一个多维的空间，那么通过用户对单一维度上的评价Score组成的坐标系X（s1,s2,s3.....，si）即可定位该用户在这个多维度空间中的位置，那么任意两个位置之间的距离Distance(X,Y)（即：欧式距离）就能在一定程度上反应了两用户兴趣的相似程度。

> 就其意义而言，欧氏距离越小，两个用户相似度就越大，欧氏距离越大，两个用户相似度就越小。

5.2、基于夹角余弦相似度算法



计算夹角，并得出夹角对应的余弦值，此余弦值就可以用来表征，这两个向量的相似性。夹角越小，余弦值越接近于1，它们的方向更加吻合，则越相似。

计算公式：

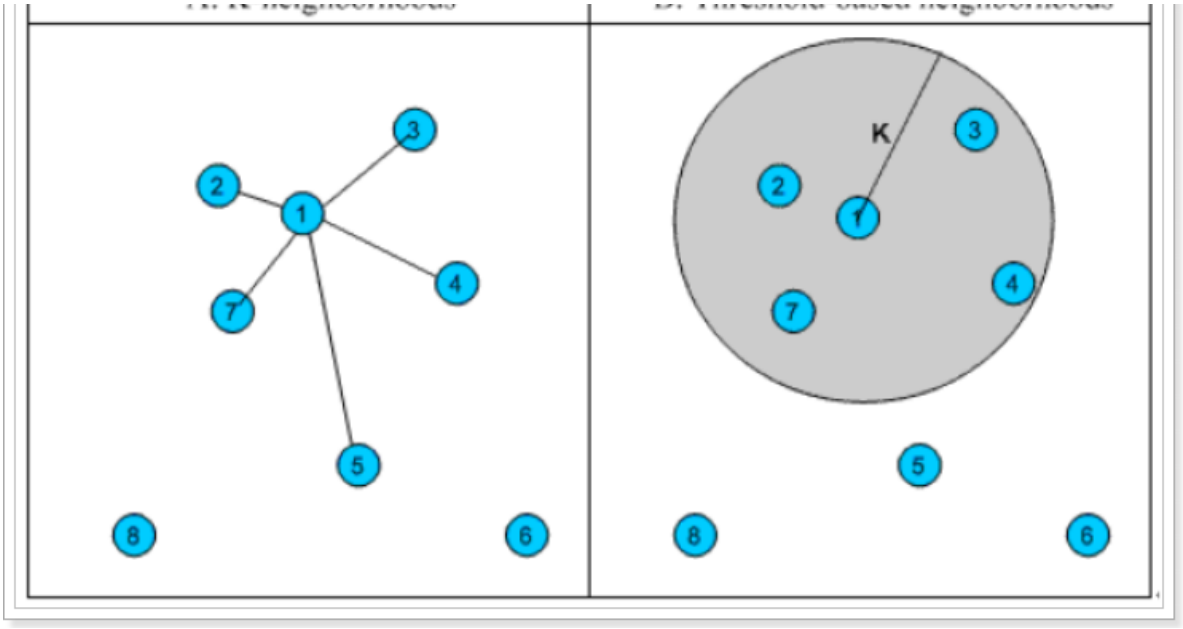
$$T(x,y)=\frac{x \bullet y}{\|x\|^2 \times \|y\|^2}=\frac{\sum x_i y_i}{\sqrt{\sum x_i^2}\sqrt{\sum y_i^2}}$$

6、最近邻域

通过相似度计算，可以计算出邻居，问题来了，我们如果选取出几个邻居作为参考，进行推荐呢？

通常有2种方式：

- 固定数量的邻居：K-neighborhoods
- 基于相似度门槛的邻居：Threshold-based neighborhoods

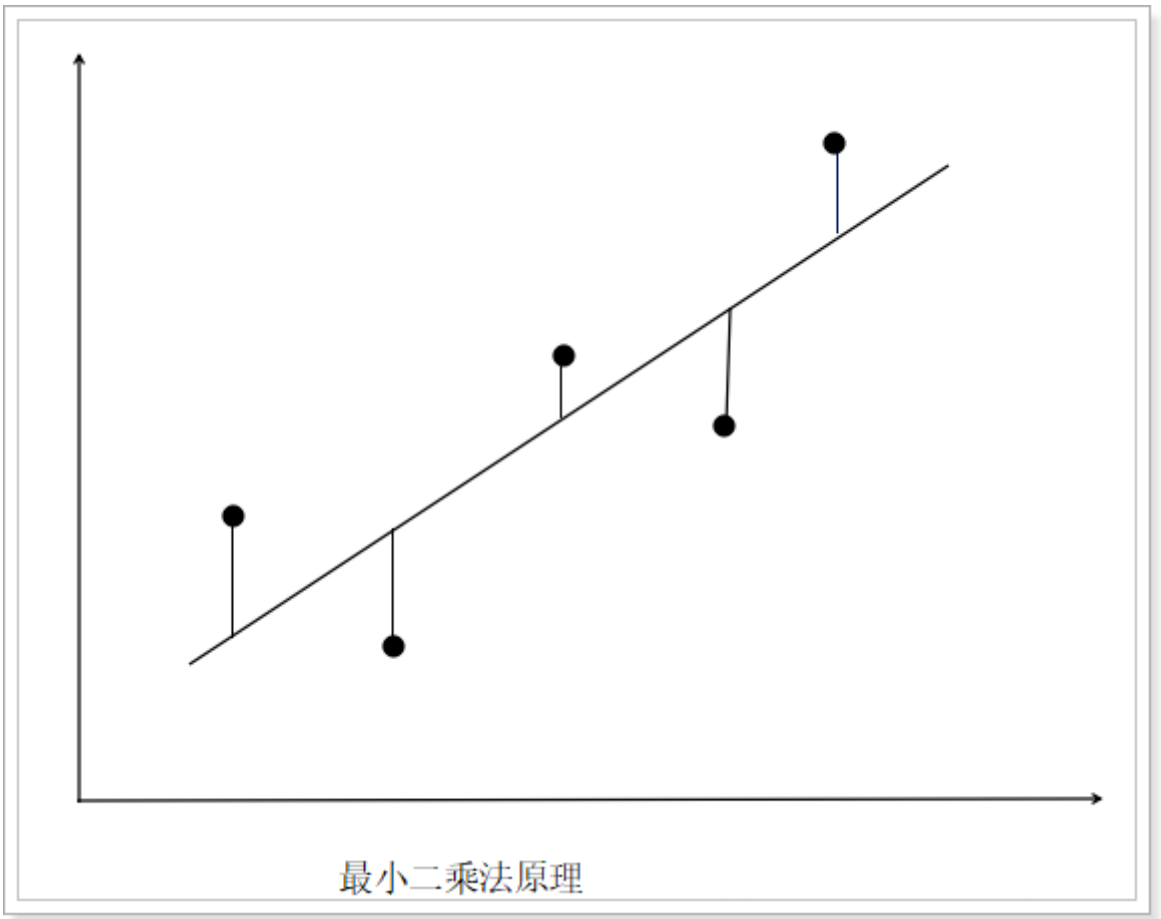


7、交替最小二乘法

交替最小二乘法（ALS）是统计分析中最常用的逼近计算的一种算法，其交替计算结果使得最终结果尽可能地逼近真实结果。而ALS的基础是最小二乘法（LS算法），LS算法是一种常用的机器学习算法，它通过最小化误差的平方和寻找数据的最佳函数匹配。利用最小二乘法可以简便的求得未知的数据，并使得这些求得的数据与实际数据之间误差的平法和为最小。

7.1、最小二乘法

以一个变量为例，在二维空间中最小二乘法的原理图如下：



若干个点依次分布在向量空间中，如果希望找出一条直线和这些点达到最佳匹配，那么最简单的一个方法就是希望这些点到直线的距离最小，则可得出最小二乘法的公式如下：

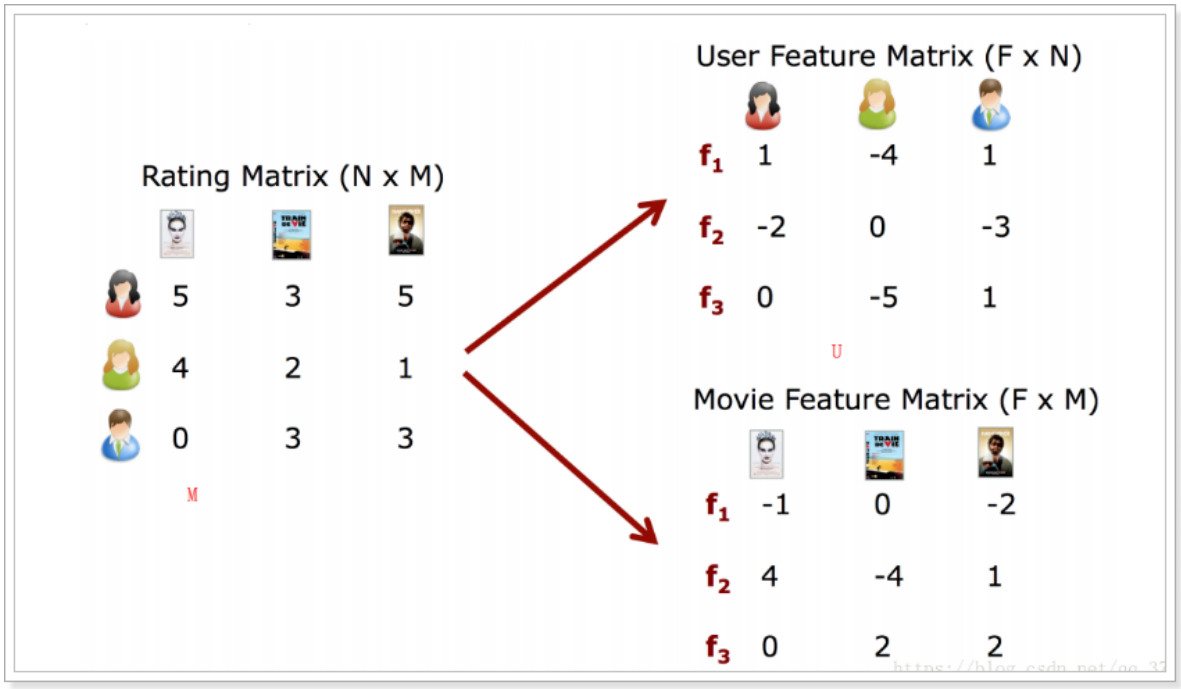
$$f(x) = ax + b$$

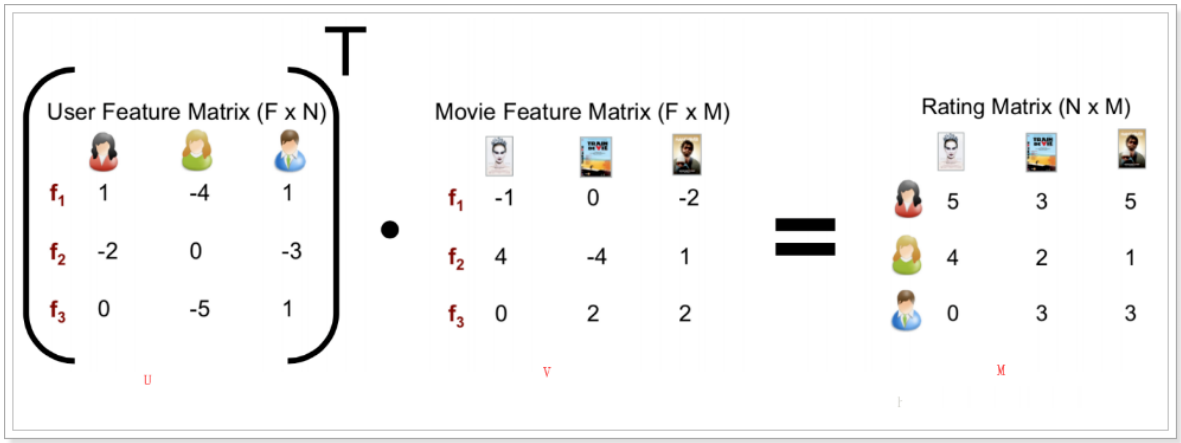
$$\delta = \sum (f(x_i) - y_i)^2$$

这里的f(x)是直接的拟合公式，也是所求的目标函数，在这里希望各个点到直接的值最小，因此第二个公式就是求所有点到该直接的距离，我们可以微分求得其最小值。

7.2、交替最小二乘法

以用户，商品为例说明，一个基于用户名，物品表的用户评分矩阵可以被分解成2个较为小型化的矩阵，即M=U的转置矩阵*V。



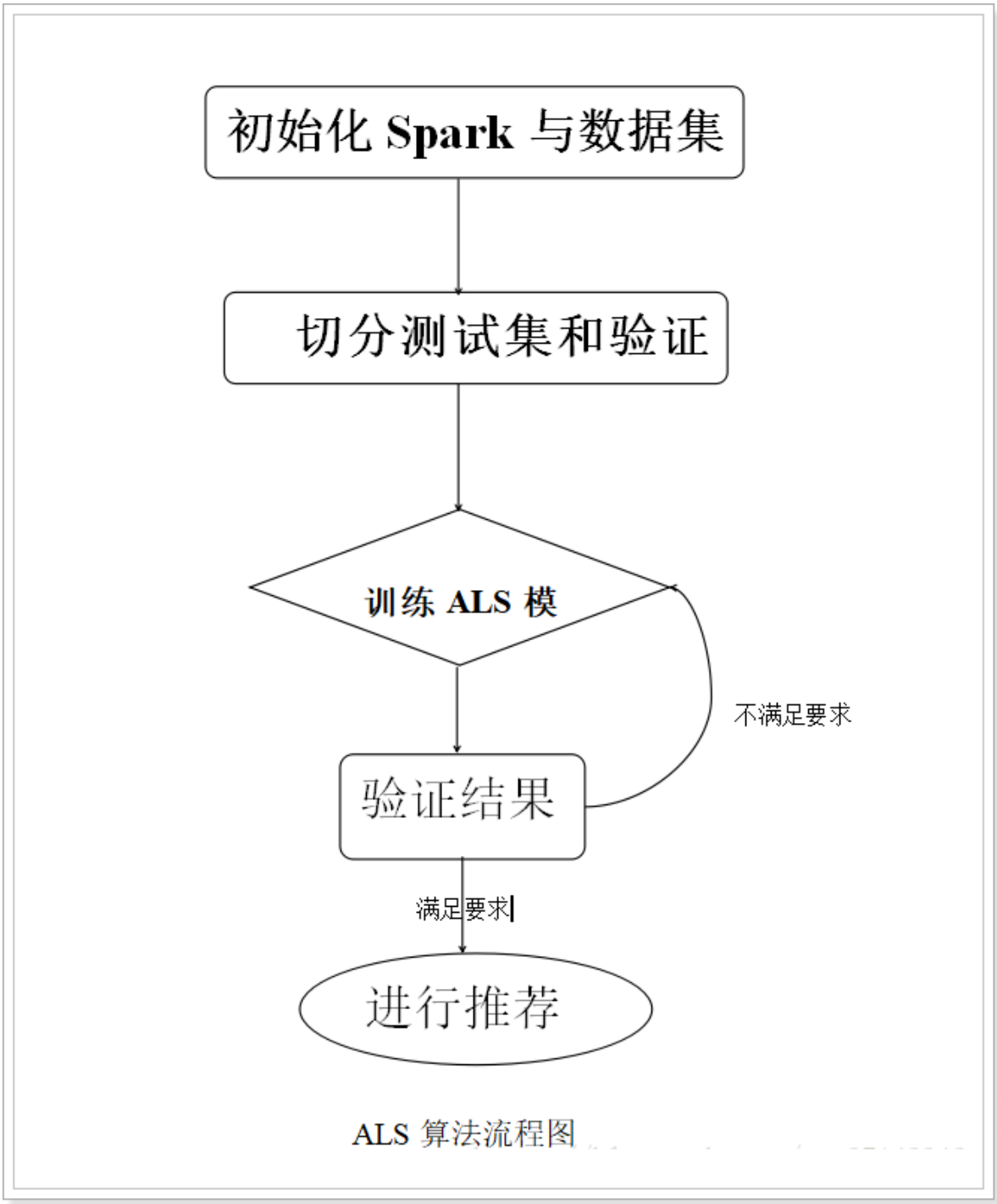


在这里U和V分别表示 用户和物品的矩阵，在MLlib的ALS算法中，首先会对U或者V矩阵随机生成，之后固定某一个特定的对象，去求取另一个未随机化的矩阵对象。

之后利用求取的矩阵对象去求随机化矩阵对象。最后两个对象相互迭代计算，求取与实际矩阵差异达到程序设定的最小阀值位置。

通俗的说就是先固定U矩阵，求取V，然后再固定V矩阵再求取U矩阵，一直这样交替迭代计算直到误差达到一定的阀值条件或者达到迭代次数的上限。

7.3、ALS算法流程



ALS 算法流程图

7.4、ALS算法实战

7.4.1、数据说明

示例数据是用户、电影、评分的数据，其中用户有943人，电影有1682部，评价有100000条。文件在资料中。

```
196 242 3   881250949
186 302 3   891717742
```

```
22  377 1   878887116
244 51  2   880606923
166 346 1   886397596
298 474 4   884182806
115 265 2   881171488
253 465 5   891628467
305 451 3   886324817
6   86  3   883603013
62  257 2   879372434
286 1014    5   879781125
200 222 5   876042340
210 40  3   891035994
.................
```

7.4.2、数据建模

ALS算法的第二步就是数据建模，其实在MLlib算法库中有可以直接使用的训练算法，ALS.tran方法源码如下：

```
 def train(
      ratings: RDD[Rating],        //需要训练的数据集
      rank: Int,                   //模型中隐藏因子数，rank一般选在8到20之间
      iterations: Int,             //算法中迭代次数，一般10次即可
      lambda: Double,              //ALS中的正则化参数，一般设置0.01
      blocks: Int,                 //并行计算的block数(-1为自动配置)
      alpha: Double,               //ALS隐式反馈变化率用于控制每次拟合修正的幅度
      seed: Long                   //加载矩阵的随机数
    ): MatrixFactorizationModel = {
    new ALS(blocks, blocks, rank, iterations, lambda, true, alpha,
seed).run(ratings)
  }
```

7.4.3、实战

```
package cn.itcast.spark;

import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.mllib.recommendation.ALS;
import org.apache.spark.mllib.recommendation.MatrixFactorizationModel;
import org.apache.spark.mllib.recommendation.Rating;

public class MyRecommend {

    public static void main(String[] args) {
        SparkConf sparkConf = new SparkConf()
                .setAppName("MyRecommend")
                .setMaster("local[*]");
        JavaSparkContext jsc = new JavaSparkContext(sparkConf);
        jsc.setLogLevel("WARN"); //设置日志级别

        JavaRDD<String> rawData = jsc.textFile("F://ml-100k//u.data");//设置
数据集文件
        JavaRDD<String[]> rawRatings = rawData.map(v1 -> v1.split("\t"));//
将数据按照\t分割
```

```
        //转化为Rating结构，参数分别为：用户id，商品id，评分
        JavaRDD<Rating> ratings = rawRatings.map(v1 -> new
Rating(Integer.valueOf(v1[0]), Integer.valueOf(v1[1]),
Double.valueOf(v1[2])));

        //设置训练模型
        MatrixFactorizationModel model = ALS.train(ratings.rdd(), 8, 10,
0.01);

        //为789用户推荐10个商品
        Rating[] recommendProducts = model.recommendProducts(789, 10);

        //打印推荐结果
        for (Rating rating : recommendProducts) {
            System.out.println(rating.user() + "->" + rating.product()+": "
+ rating.rating());
        }
    }
}

```

7.4.4、优化改进

在上面的实战中，rank、iterations、lambda参数都是写死的，根据不同环境和数据集需要作出调整，所以我们需要计算中最佳的参数，才能得到最佳的训练集。

```
package cn.itcast.spark;

import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.mllib.recommendation.ALS;
import org.apache.spark.mllib.recommendation.MatrixFactorizationModel;
import org.apache.spark.mllib.recommendation.Rating;
import scala.Tuple2;

import java.util.List;

public class MyRecommend2 {

    public static void main(String[] args) {
        SparkConf sparkConf = new SparkConf()
                .setAppName("MyRecommend")
                .setMaster("local[*]");
        JavaSparkContext jsc = new JavaSparkContext(sparkConf);
        jsc.setLogLevel("WARN"); //设置日志级别

        JavaRDD<String> rawData = jsc.textFile("F://ml-100k//u.data");//设
置数据集文件
        JavaRDD<String[]> rawRatings = rawData.map(v1 ->
v1.split("\t"));//将数据按照\t分割
        //装载样本评分数据，其中最后一列Timestamp取除10的余数作为key，Rating为值，
即(Int，Rating)
        JavaPairRDD<Long, Rating> ratings = rawRatings.mapToPair(v1 -> {
            Rating rating = new Rating(Integer.valueOf(v1[0]),
Integer.valueOf(v1[1]), Double.valueOf(v1[2]));
```

```
            return new Tuple2<>(Long.valueOf(v1[3]) % 10, rating);
        });

        //装载电影目录对照表(电影ID->电影标题)
        List<Tuple2> movies = jsc.textFile("F://ml-100k//u.item").map(v1 -> {
            String[] ss = v1.split("\\|");
            return new Tuple2(ss[0], ss[1]);
        }).collect();

        //统计有用户数量和电影数量以及用户对电影的评分数目
        Long numRatings = ratings.count();
        Long numUsers = ratings.map(v1 -> ((Rating) v1._2()).user()).distinct().count();
        Long numMovies = ratings.map(v1 -> ((Rating) v1._2()).product()).distinct().count();
        System.out.println("用户：" + numUsers + "电影：" + numMovies + "评论：" + numRatings);

        //将样本评分表以key值切分成3个部分，分别用于训练 (60%，并加入用户评分)， 校验 (20%)， and 测试 (20%)
        //该数据在计算过程中要多次应用到，所以cache到内存

        Integer numPartitions = 4; // 分区数
        // 训练集
        JavaRDD<Rating> training = ratings
                .filter(v -> v._1() < 6)
                .values()
                .repartition(numPartitions)
                .cache();

        // 校验集
        JavaRDD<Rating> validation = ratings
                .filter(v -> v._1() >= 6 && v._1() < 8)
                .values()
                .repartition(numPartitions).cache();

        // 测试集
        JavaRDD<Rating> test = ratings
                .filter(v -> v._1() >= 8)
                .values()
                .cache();

        Long numTraining = training.count();
        Long numValidation = validation.count();
        Long numTest = test.count();
        System.out.println("训练集：" + numTraining + " 校验集：" + numValidation + " 测试集：" + numTest);

        //训练不同参数下的模型，并在校验集中验证，获取最佳参数下的模
        int[] ranks = new int[]{10, 11, 12};
//        double[] lambdas = new double[]{0.01, 0.03, 0.1, 0.3, 1, 3};
        double[] lambdas = new double[]{0.01};
//        int[] numIters = new int[]{8, 9, 10, 11, 12, 13, 14, 15};
        int[] numIters = new int[]{8, 9, 10};

        MatrixFactorizationModel bestModel = null;
        double bestValidationRmse = Double.MAX_VALUE;
```

```
        int bestRank = 0;
        double bestLambda = -0.01;
        int bestNumIter = 0;

        for (int rank : ranks) {
            for (int numIter : numIters) {
                for (double lambda : lambdas) {
                    MatrixFactorizationModel model =
ALS.train(training.rdd(), rank, numIter, lambda);
                    Double validationRmse = computeRmse(model, validation,
numValidation);
                    System.out.println("RMSE(校验集) = " + validationRmse +
", rank = " + rank + ", lambda = " + lambda + ", numIter = " + numIter);

                    if (validationRmse < bestValidationRmse) {
                        bestModel = model;
                        bestValidationRmse = validationRmse;
                        bestRank = rank;
                        bestLambda = lambda;
                        bestNumIter = numIter;
                    }
                }
            }
        }



        double testRmse = computeRmse(bestModel, test, numTest);
        System.out.println("测试数据集在 最佳训练模型 rank = " + bestRank + ",
lambda = " + bestLambda + ", numIter = " + bestNumIter + ", RMSE = " +
testRmse);

        // 计算均值
        Double meanRating = training.union(validation).mapToDouble(v ->
v.rating()).mean();

        // 计算标准误差值
        Double baselineRmse = Math.sqrt(test.map(v -> (meanRating -
v.rating()) * (meanRating - v.rating())).reduce((v1, v2) -> (v1 + v2) /
numTest));

        // 计算准确率提升了多少
        double improvement = (baselineRmse - testRmse) / baselineRmse *
100;

        System.out.println("最佳训练模型的准确率提升了：" +
String.format("%.2f", improvement) + "%.");


        // 构建最佳训练模型
        bestModel = ALS.train(ratings.values().rdd(), bestRank,
bestNumIter, bestLambda);

        Rating[] recommendProducts = bestModel.recommendProducts(789, 10);
        //打印推荐结果
        for (Rating rating : recommendProducts) {
            System.out.println(rating.user() + "->" + rating.product()+":
" + rating.rating());
```

```
        }

    }

    /**
     * 校验集预测数据和实际数据之间的均方根误差
     **/
    public static Double computeRmse(MatrixFactorizationModel model, JavaRDD<Rating> data, Long n) {
        // 进行预测
        JavaRDD<Rating> predictions = model.predict(data.mapToPair(v -> new Tuple2<>(v.user(), v.product())));

        JavaRDD<Tuple2<Double, Double>> predictionsAndRatings = predictions
                .mapToPair(v -> new Tuple2<>(new Tuple2<>(v.user(), v.product()), v.rating()))
                .join(data.mapToPair(v -> new Tuple2<>(new Tuple2<>(v.user(), v.product()), v.rating()))).values();

        Double reduce = predictionsAndRatings.map(v -> (v._1 - v._2) * (v._1 - v._2))
                .reduce((v1, v2) -> (v1 + v2) / n);
        //正平方根
        return Math.sqrt(reduce);
    }
}

```

运行结果：

```
用户：943电影：1682评论：100000
训练集：60024 校验集：20435 测试集：19541

RMSE(校验集) = 1.9042189155615398E-5, rank = 10, lambda = 0.01, numIter = 8
RMSE(校验集) = 4.241470253244084E-5, rank = 10, lambda = 0.01, numIter = 9
RMSE(校验集) = 1.4449138813397543E-5, rank = 10, lambda = 0.01, numIter = 10
RMSE(校验集) = 6.082115231352825E-5, rank = 11, lambda = 0.01, numIter = 8
RMSE(校验集) = 7.053700942035736E-5, rank = 11, lambda = 0.01, numIter = 9
RMSE(校验集) = 2.6551033361283012E-5, rank = 11, lambda = 0.01, numIter = 10
RMSE(校验集) = 3.31869548209156E-5, rank = 12, lambda = 0.01, numIter = 8
RMSE(校验集) = 7.16881674580905E-5, rank = 12, lambda = 0.01, numIter = 9
RMSE(校验集) = 4.8896811754448735E-5, rank = 12, lambda = 0.01, numIter = 10
测试数据集在 最佳训练模型 rank = 10, lambda = 0.01, numIter = 10, RMSE = 1.4925418704176325E-5

最佳训练模型的准确率提升了：81.34%.

789->1598: 9.396627380043618
789->1368: 8.14482648449507
789->1466: 7.38882286732828
789->1462: 6.81871292883074
789->1059: 6.795506701087147
789->337: 6.780524189917936
789->267: 6.772379061916583
789->1184: 6.688179558303117
789->906: 6.528331428606722
```

```
789->854: 6.466069010345999

```

从运行效果来看：

- 总共有943个用户，1682个电影（已经去重），100000条评分数据；
- 如程序，我们把所有数据分为三部分：
  - 60%用于训练、20%用于校验、20%用于测试模型；
- 模型在不同参数下的均方根误差（RMSE）值，以及对应的参数，最优的参数选择均方根误差（RMSE）最小的参数值---即最优参数模型建立；
- 使用20%的测试模型数据来测试模型的好坏，也就是均方根误差（RMSE），在最优参数模型基础上提升了81.34%的准确率。