2012 暑假大作业项目说明

SNSMC-“Item-based 协同过滤”算法

万志程

5090379104

一、背景介绍

1. 目标场景：

如今微博已经融入到人们的日常生活中，截至 2012 年 2 月 28 日，新浪微博注册用户数突破 3 亿，每日发微博量超过 1 亿，如此大的数据量，使得浏览微博的人很快迷失在微博的海洋中。为了能迅速找到每个人喜欢的东西，我们需要对新浪微博上 的一定数量的特定人群进行个性化推荐，使他们能快速定位自己喜欢的内容，而不必去手动查找。另外，推荐信息还可以作为投放广告，电子商务等活动的重要依据。这其中，推荐内容包括微博帐号、内容、活动等。

2. 使用工具：

新浪微博官方 sdk（Java 版本）、redis 服务器、redis 的 Java 客户端 jedis、redis 的 c 客户端 hiredis、ketama 哈希算法代码。从微博上抓到的数据先转换成自定义的格式进行中间数据读取和保存，然后再把合并以后的数据存入 redis 数据库当中。主程序则是用 c 语言从数据库取数据，来进行处理和分析的。

3. 预期效果：

输入某个特定用户的 ID 和所要求用户量， 输出该用户所要求数据量的 follower 或者 friends 中所有人的推荐项目。

二、系统设计

1. 算法

所采用的算法共分为三部分，分别是提取 item 时用的提取算法、计算推荐时用的 Item-based 协同过滤的算法和数据库 sharding 时的 ketama 哈希算法。

1.1 提取 Item：

1.1 提取数据的算法：抽取每个用户最近 50 条微博内容当中“@”后、“#”包围、“【”包围、单引号包围以及“http://t.cn/”后的内容。

1.2 原因：“@”通常代表的认识的人或者帐号，“#”通常代表感兴趣的话题，“【”和单引号代表感兴趣内容的标题，而“http://t.cn/”代表感兴趣的链接。

1.2 推荐算法：

如今已经进入了一个数据爆炸的时代，随着 Web 2.0 的发展， Web 已经变成数据分享的平台，那么，如何让人们在海量的数据中想要找到他们需要的信息将变得越来越难。

在这样的情形下，搜索引擎（Google，Bing，百度等等）成为大家快速找到目标信息的最好途径。在用户对自己需求相对明确的时候，用搜索引擎很方便的通过关键字搜索很快的找到自己需要的信息。但搜索引擎并不能完全满足用户对信息发现的需求，那是因为在很多情况下，用户其实并不明确自己的需要，或者他们的需求很难用简单的关键字来表述。又或者他们需要更加符合他们个人口味和喜好的结果，因此出现了推荐系统，与搜索引擎对应，大家也习惯称它为推荐引擎。随着推荐引擎的出现，用户获取信息的方式从简单的目标明确的数据的搜索转换到更高级更符合人们使用习惯的信息发现。

如今，随着推荐技术的不断发展，推荐引擎已经在电子商务 (E-commerce，例如 Amazon，当当网) 和一些基于 social 的社会化站点 (包括音乐，电影和图书分享，例如豆瓣，Mtime 等) 都取得很大的成功。这也进一步的说明了，Web2.0 环境下，在面对海量的数据，用户需要这种更加智能的，更加了解他们需求，口味和喜好的信息发现机制。

根据推荐物品或内容的元数据，发现物品或者内容的相关性，这种被称为基于内容的推荐（Item-based 的协同过滤），该算法将所有用户对所有 Item 的评分（未评的为 0 分）作为输入。然后利用向量余弦定理计算任意两个 Item 之间的相似度。最后再根据已有评分的 Item 和未评分 Item 的相似度利用加权平均计算未评分 Item 的估计评分，估计评分最高的作为推荐对象。

基于项目的协同过滤推荐的基本原理是使用所有用户对物品或者信息的偏好，发现物品和物品之间的相似度，然后根据用户的历史偏好信息，将类似的物品推荐给用户，上图很好的诠释了它的基本原理。

假设用户 A 喜欢物品 A 和物品 C，用户 B 喜欢物品 A，物品 B 和物品 C，用户 C 喜欢物品 A，从这些用户的历史喜好可以分析出物品 A 和物品 C 时比较类似的，喜欢物品 A 的人都喜欢物品 C，基于这个数据可以推断用户 C 很有可能也喜欢物品 C，所以系统会将物品 C 推荐给用户 C。

1.3 ketama 算法：

ketama 是一致的散列算法的实现，这意味着可以从 memcached 的 pool 添加或删除服务器，而不需要重映射所有键。
下面是它的工作原理：
*读取服务器列表（如：1.2.3.4:11211，5.6.7.8:11211，9.8.7.6:11211）；
*每个服务器字符串哈希成几个（100-200）无符号整数;
*从概念上讲，这些数字被放置在一个称为 continuum 的环上。 （想象一个钟面，从 0 到 2 ^ 32）；
*每个数字链接到服务器上，服务器出现在 continuum 的几个点上，由每一个数字散列。
*要映射一个键到服务器，哈希键映射到一个单一的无符号整数，并找到下一个最大的连续号码。连接到这个数字的服务器就是对应该键的服务器;
*如果哈希键的值接近 2 ^ 32，没有点在 continuum 上大于你的哈希值，则返回的第一台服务器。
如果添加或删除列表中的服务器，只有很少部分的键最终会映射到不同的服务器上。

2. 数据结构

2.1 数据抽取

数据抽取所用数据结构都是新浪微博 sdk 用提供的各种数据类，如 Friendships、UserWapper、Users、Timeline、StatusWapper 等等

2.2 中间结果

存储中间结果到文件中时用的是 jedis 提供的一些数据类，如 Jedis、Tuple 等等。

2.4 redis

redis 中存储的数据结构用的是 Sorted-Set，用户 ID 作为 key，评分作为 score，Item 名称作为 value。

2.5 处理分析

计算推荐评分时所用数据结构主要是二维数组。用它来存储评分的二维矩 阵，其次还有一些数组用来存储所有用户和所有 Item。

2.6 sharding

服务器 sharding 时所用的数据结构是一些自定的结构以及他们的数组，如 sync_t、redis_srv 等等。

3. 模块功能

3.1 获取微博数据

通过 weibo sdk 的 java 版本获取特定用户的一定数量的朋友及其最近一段时间内发的 50 条微博，在 FetchData.java 的 FetchData 类的构造函数中，用新浪微博 sdk 实现。

3.2 提取 item

通过提取微博内容的关键词来获得所需的 Item，在 FetchData.java 中的 Retrieval 函数实现。

3.3 读取历史数据

从中间结果文件当中读取原来保存的结果数据，在 FetchData.java 的 FecthHistoryData 函数中实现。

3.4 存储中间数据

将获取到的 item 数据存入中间文件当中。在 FetchData.java 的 FetchData 类的构造函数及 SaveBlogsToRedisByUid 函数中实现，把用户和 Item 信息存入中间文件”Result.txt”。

3.5 存入 redis

将整合后的数据同时存入 redis 服务器当中，在 FetchData.java 的 FetchData 类的构造函数及 SaveBlogsToRedisByUid 函数中实现。

3.6 构造评分矩阵

从 redis 中取出数据来构造评分矩阵，用户为行，Item 为列，讲评分依次输入，在 ItemRecommend.c 中的 generate_rating 系列函数中实现。

3.7 计算相似度

通过计算余弦夹角的方法计算任意两个 item 相似度，并将结果存入相似度矩阵，在 ItemRecommend.c 中的 calculate_simularity 函数中实现。

3.8 计算估计评分和推荐 item

通过加权平均的方法计算估计的评分，并选出评分最高的 Item，在 ItemRecommend.c 中的 generate_recommend 系列函数中实现。

3.9 多线程

多线程主要应用在构造评分矩阵和计算推荐项目时用到，将每个用户的计算任务独立为一个线程。在 ItemRecommend 的 generate_rating_multi_thread 函数和 generate_all_recommend_multi_thread 函数中实现。

3.10 sharding

利用现有 ketama 算法接口将一个 rendis 实例分为多个（2~4），并将用户数据平均放入这几个实例当中，分别进行计算。在 UserInsert.c 当中将用户数据通过 ketama 哈希算法放入到几个 redis 实例中，在 ItemRecommend.c 当中 geretate_rating_table_sharded_multi_thread 函数当中使用这些数据。

三、系统实现

1. 自写代码

FetchData.java、ItemRecommend.c、大部分 UserInsert.c。

2. 现成代码

ketama 算法、UserInsert.c 中部分线程同步代码。

3. 对应模块和功能

FetchData.java 包括获取微博数据、提取 Item、 读取历史数据、存储中间数据和将数据存入 redis 功能模块，对应关系见上“模块功能”部分。
ItemRecommend.c 中包括 构造评分矩阵、 计算 Item 之间的相似度、 计算估计评分和推荐 Item 功能模块。UserInsert.c 中包括 sharding 功能。具体对应关系见上“模块功能”部分。

4. 程序编译方法

(1). FetchData.java:
javac FetchData.java

(2). ItemRecommend.c:
gcc -Wall -pthread -std=c99 -D_GNU_SOURCE "ItemRecommend.c" "hiredis.c" "comm.c" "sds.c" "net.c" -lm -o "ItemRecommend"

(3). UserInsert.c:
gcc -Wall -pthread -std=c99 -D_GNU_SOURCE "UserInsert.c" "comm.c" "hiredis.c" "md5.c" "sds.c" "net.c" "ketama.c" -lm -o "UserInsert"

5. 程序运行方法

(1). FetchData.java:
java FetchData

(2). ItemRecommend.c:
./ItemRecommend

(3). UserInsert.c:
./UserInsert

四、系统测试

1. 系统环境

OS： linux mint 13(maya) 64bit;
Core： Intel® Core™2 Duo CPU P7450 @ 2.13GHz × 2 ;

2. 输入/输出

输入：(access token)； 1785188851； 50
输出：recommendation result:
(No. id name
est_score item_name)

3) 1445592647 万小 e 勇气
2.000000 美食和旅行

3. 性能

以上例输入为例。
(1). 单线程

功能模块	构造评分表	计算相似度	计算估计分数	总时间
Time(s)	3.98	6.88	3.27	14.15

(2). 多线程

功能模块	构造评分表	计算相似度	计算估计分数	总时间
Time(s)	17.15	6.49	5.43	29.00

(3). Sharding

功能模块	构造评分表	计算相似度	计算估计分数	总时间
Time(s)	0.27	5.49	4.7	10.18

将用户数调为 100 得到如下结果。
(1). 单线程

功能模块	构造评分表	计算相似度	计算估计分数	总时间
Time(s)	7.21	10.23	7.23	26.32

(2). 多线程

功能模块	构造评分表	计算相似度	计算估计分数	总时间
Time(s)	35.56	10.73	13.23	60.34

(3). Sharding

功能模块	构造评分表	计算相似度	计算估计分数	总时间
Time(s)	2.48	11.93	10.20	24.92

4. 测试结论

(1). 多核环境下数据处理性能与单核不同，因为线程在多核间调度的开销，多线程的效率不一定比单线程高；

(2). 多实例多线程的执行效率要比单实例多线程的效率高很多，原因是实现了并行计算；

(3). 多线程多实例的执行效率也比单实例单线程要高，原因同上；

(4). 实例分得越多总体执行效率越高；

(5). 数据量越大效率越低，且单实例多线程的效率下降幅度最大，可能的原因是数据变多引起线程增多，增加线程切换的额外开销增大。

5. 性能瓶颈

因为只分了四个 redis 实例，所以性能提升不是很明显。同时因为该算法局部依赖于整体的关系，每更新一次局部数据就需要重新构造整张评分表，重新计算所有 Item 的相似度，所以实时性的效率不太高。

五、收获与体会

1. 收获

学到了新浪微博的开发技术，对 c 语言的 pthread 多线程编程技术有比较详细的了解，对数据挖掘中的 Item-based 推荐算法有了深入了解，掌握了 redis 数据存储的各种技术。

2. 体会

数据处理性能的优劣关系到很多方面，包括数据量大小、实时性、线程调度、以及一些是否有一些特殊的优化机制（如 pipeline）等等。